//Neverland Oracle PR Review

Prepared For: Neverland
17 February 2026

About Octane

Founded in 2023 and headquartered in San Francisco, Octane delivers a truly holistic security solution through continuous code reviews that protect protocols throughout their entire development lifecycle. We combine software-first vulnerability analysis with world-class manual security talent, enabling teams to ship faster while maintaining the highest security standards. Our clients including Circle (USDC), Uniswap, Avalanche, Plume, and Sophon trust us to protect $5.8B+ in assets because we catch vulnerabilities that traditional assessments miss.

Clients choose Octane for:

- Our software-first approach: We've built a platform trained on thousands of real-world vulnerabilities that consistently identifies complex exploit patterns traditional methods miss. Every commit and pull request receives automatic review through native CI/CD integration, with vulnerabilities identified through deep dependency analysis and exploit path tracing.

- Our novel research: Octane's security researchers who've collectively earned over $1M in bug bounties provide comprehensive manual reviews on an as-needed basis, maximizing code security and developing further insights for automated protection.

- Our proven track record preventing exploits: We've discovered 100+ vulnerabilities across 86M+ lines of code that were missed in traditional security reviews. We prevented exploits similar to Rikkei Finance ($1.1M), caught complex slippage bugs in Uniswap integrations, identified oracle mispricing vulnerabilities, and found assembly errors that evaded static analyzers. Our platform adapts to your team's preferences, dramatically reducing false positives as the platform learns your codebase while providing exact code modifications and exploit scenarios for immediate remediation.

- Our commitment to ecosystem security: We partner directly with blockchain foundations to establish network-wide security baselines. Integration requires no code changes and takes minutes to deploy, after which teams receive continuous protection with rapid response. We offer flexible engagement from basic reviews to enterprise coverage with 24/7 support and formal security reports, supporting all EVM-compatible chains with developing Rust capabilities.

Through our holistic approach, we transform security from a development bottleneck into an enabler of velocity and confidence. We live by iterative development and in August FY25 alone, we boosted accuracy on challenging vulnerability classes by 31% and increased our detection coverage by 12.7%. To learn how continuous security accelerates development while protecting users, visit octane.security.

Engagement Overview

Neverland requested a review of PR 30 in the Neverland-Money/neverland-contracts repository at commit `34e8189`. This PR introduces two oracle adapters to provide Chainlink-compatible price feeds for yield-bearing assets for use with Aave V3-compatible lending markets.

From February 3 to February 5, 2026, one Octane security researcher reviewed the new oracle adapter implementations. The review focused on identifying any integration issues with the `AaveOracle` or the underlying Chainlink price feeds and ERC4626 vaults, as well as any unhandled edge cases that could result in corrupt data or bypassed access controls.

This review identified 15 issues. Four low-severity issues that highlight some inconsistencies in how data is handled between functions and 11 informational-severity issues that generally relate to code cleanliness or redundant code that can be removed. Only one low-severity issue affected the `latestAnswer` function, which is the key component for integrating with `AaveOracle`. The `YieldBearingOracleAdapter`'s implementation artificially clamps the ratio used to calculate the price if it exceeds the configured maximum growth rate. This contradicts the convention followed throughout both contracts that functions should revert/return 0 when bad data is encountered (the PR description also notes that fallback logic is the responsibility of the consuming protocols).

Scope

The scope of this review was limited to the oracle adapter contracts in PR 30 of the Neverland-Money/neverland-contracts repository up to commit `34e8189`.

The following smart contracts were in scope of the audit:

- `src/oracles/RatioOracleAggregator.sol`
- `src/oracles/YieldBearingOracleAdapter.sol`
- `src/libraries/OracleLib.sol`
- `src/interfaces/IYieldBearingOracleAdapter.sol`
- `src/interfaces/IRatioOracleAggregator.sol`

On February 16, 2026, we reviewed the fixes for the issues presented in this report. These were included in the same PR as of commit `47354fc`. All of the reported issues have been fixed.

Findings

ID	Title	Status
L-1	Growth cap provides authoritative incorrect pricing instead of triggering fallback	FIXED
L-2	Inconsistent error handling across legacy AggregatorInterface functions	FIXED
L-3	getAnswer and getTimestamp return misleading data for non-latest rounds	FIXED
L-4	Inconsistency between latestAnswer, latestRound, and latestTimestamp return values	FIXED
I-1	Missing AggregatorV3Interface inheritance in IYieldBearingOracleAdapter	FIXED
I-2	_getDecimals silently falls back, masking aggregator misconfiguration	FIXED
I-3	isCapped() conflates not capped with error conditions	FIXED
I-4	Reliance on deprecated answeredInRound Chainlink field	FIXED
I-5	Redundant validation checks duplicated across functions	FIXED
I-6	Dead code branch in _rolloverEffectiveSnapshot	FIXED
I-7	Redundant external calls in getRoundData implementations	FIXED
I-8	Magic number for maximum decimals should be a shared constant	FIXED
I-9	Parameter order in _ensureComplete does not match latestRoundData return order	FIXED
I-10	Confusing snapshot state terminology and misleading function names	FIXED
I-11	Reverting function used in view path where non-reverting variant exists	FIXED

L-1 Growth cap provides authoritative incorrect pricing instead of triggering fallback

Description

When the actual vault ratio from `convertToAssets()` exceeds the growth cap, `_computeSharesPriceSafe` silently clamps the ratio to the maximum allowed value and returns an authoritative price:

```
// YieldBearingOracleAdapter.sol:367-371
if (_hasEffectiveSnapshot()) {
    uint256 maxAllowedRatio = _calculateMaxRatioView();
    if (currentRatio > maxAllowedRatio) {
        currentRatio = maxAllowedRatio;
    }
}
```

This means if the vault's growth rate legitimately exceeds the configured cap (e.g., a high-yield period or a cap set too conservatively), the adapter returns a price that is knowingly incorrect rather than returning 0 to trigger `AaveOracle`'s fallback mechanism.

This design prioritizes availability over correctness, when availability is also provided through the oracle fallback mechanism. A consumer receiving a capped price has no way to distinguish it from a genuine price, and the capped value could be materially different from the real vault value. In a lending context, this could cause liquidations based on stale/incorrect pricing or prevent liquidations that should occur.

Recommendation

Consider returning 0 when the ratio exceeds the cap.

Resolution

`_computeSharesPriceSafe` now returns 0 when the ratio exceeds the cap, triggering the fallback mechanism.

L-2 Inconsistent error handling across legacy `AggregatorInterface` functions

Description

The legacy `AggregatorInterface` functions in both adapter contracts handle errors inconsistently. `latestAnswer()`, `getAnswer()`, and `getTimestamp()` all return 0 on failure, but `latestTimestamp()` reverts:

```
// RatioOracleAggregator.sol:154-163
function latestTimestamp() external view override returns (uint256) {
    (uint80 roundId,,, uint256 ratioUpdatedAt, uint80 answeredInRound) =
ratioAggregator.latestRoundData();
    _ensureComplete(ratioUpdatedAt, roundId, answeredInRound); // reverts!

    (uint80 baseRoundId,,, uint256 baseUpdatedAt, uint80
baseAnsweredInRound) = baseAggregator.latestRoundData();
    _ensureComplete(baseUpdatedAt, baseRoundId, baseAnsweredInRound); //
reverts!

    return ratioUpdatedAt < baseUpdatedAt ? ratioUpdatedAt : baseUpdatedAt;
}
```

The same pattern exists in `YieldBearingOracleAdapter`'s `latestTimestamp()`, which reverts via `_validateFreshness` while `latestAnswer()` returns 0.

A consumer calling `latestTimestamp()` would get a revert, while calling `latestAnswer()` would get 0 for the same underlying data failure. This inconsistency could cause issues for integrators that call both functions and expect uniform error semantics.

Recommendation

Update the `latestTimestamp` implementations to use non-reverting validation functions to align error handling across all legacy interface functions.

Resolution

The legacy interface functions all use a new `_tryLatestData` function and consistently return 0 on failure.

L-3 `getAnswer` and `getTimestamp` return misleading data for non-latest rounds

Description

In `YieldBearingOracleAdapter`, `getAnswer()` and `getTimestamp()` accept arbitrary round IDs and query the base aggregator's `getRoundData()` for that round. However, they always compute the price using the current `convertToAssets()` ratio, not the ratio that was in effect at that round's timestamp:

```
// YieldBearingOracleAdapter.sol:180-189
function getAnswer(uint256 roundId) external view returns (int256) {
    if (roundId > type(uint80).max) return 0;
    try baseAggregator.getRoundData(uint80(roundId)) returns (
        uint80 resolvedRoundId, int256 answer, uint256, uint256 timestamp,
uint80 answeredInRound
    ) {
        if (!_isValidRound(answer, timestamp, resolvedRoundId,
answeredInRound)) return 0;
        return _computeSharesPriceSafe(answer);
    } catch {
        return 0;
    }
}
```

Since historical snapshot data is not stored, this computes a price mixing a historical base price with the current vault ratio. Compare with `RatioOracleAggregator.getAnswer()` which rejects non-latest rounds entirely by checking `roundId != latestRoundId`.

The interface natspec mentions that historical rounds are not supported for `getRoundData`, but `getAnswer` and `getTimestamp` do not document this limitation and do not enforce latest-only semantics.

Recommendation

Update these functions to call `this.latestAnswer()` / `this.latestTimestamp()` and verify the `roundId` matches.

Resolution

`getAnswer` and `getTimestamp` in both adapter contracts use the same pattern, delegating to `_tryLatestData` and rejecting non-latest round IDs.

L-4 Inconsistency between `latestAnswer`, `latestRound`, and `latestTimestamp` return values

Description

In `RatioOracleAggregator`, `latestTimestamp()` returns the minimum of the two feed timestamps, while `latestAnswer()` returns 0 when price computation fails. This means it is possible for `latestAnswer()` to return 0 (indicating no valid price) while `latestTimestamp()` returns a valid recent timestamp for the same underlying data:

```
// RatioOracleAggregator.sol:165-168
function latestRound() external view override returns (uint256) {
    (uint80 roundId,,,,) = ratioAggregator.latestRoundData();
    return uint256(roundId);
}
```

Additionally, `latestRound()` returns the ratio feed's round ID without validating it, while `latestTimestamp()` validates both feeds. A consumer checking `latestRound()` and `latestTimestamp()` could see valid-looking metadata for a round where `latestAnswer()` returns 0.

This issue also applies to the same functions in `YieldBearingOracleAdapter`. Though the implementations differ slightly, the recommendations remain the same.

Recommendation

Ensure consistency: if `latestAnswer()` would return 0, then `latestTimestamp()` and `latestRound()` should also signal failure. Consider having the legacy functions delegate to a shared internal computation and return only the relevant fields.

Alternatively, consider whether functions other than `latestAnswer()` are actually necessary to implement. Their implementations could be replaced with `return 0` or `revert`

`CommonChecksLibrary.NotImplemented()` to ease the maintenance burden, especially since several cannot return semantically useful data anyway.

Resolution

All of the legacy interface functions delegate to a shared `_tryLatestData` function and return 0 consistent with `latestAnswer`.

I-1 Missing `AggregatorV3Interface` inheritance in `IYieldBearingOracleAdapter`

Description

`IYieldBearingOracleAdapter` inherits from `AggregatorInterface` but not from `AggregatorV3Interface`. The implementation contract `YieldBearingOracleAdapter` implements the V3 functions but these are defined directly in the interface rather than inherited from `AggregatorV3Interface`.

Additionally, the `YieldBearingOracleAdapter` implementation's natspec uses `@inheritdoc IYieldBearingOracleAdapter` instead of from each respective interface, also in contrast with `RatioOracleAggregator`.

Recommendation

Have `IYieldBearingOracleAdapter` inherit from both `AggregatorInterface` and `AggregatorV3Interface`, and remove the duplicate function signatures from the custom interface. This ensures compatibility with consumers expecting `AggregatorV3Interface`.

Resolution

`IYIeldBearingOracleAdapter` now inherits `AggregatorV3Interface`, the natspec tags were updated, and the duplicate function signatures were removed.

I-2 `_getDecimals` silently falls back, masking aggregator misconfiguration

Description

The `_getDecimals` function in `YieldBearingOracleAdapter` catches all errors from `AggregatorV3Interface.decimals()` and returns a default of 8:

```
// YieldBearingOracleAdapter.sol:534-540
function _getDecimals(address aggregator) internal view returns (uint8) {
    try AggregatorV3Interface(aggregator).decimals() returns (uint8 dec) {
        return dec;
    } catch {
        return 8;
    }
}
```

This is only used in the constructor for the base aggregator. If a misconfigured address is passed, the constructor may silently assume 8 decimals and proceed. A misconfigured address could still have a successful `latestRoundData()` call in some edge cases, leading to incorrect price scaling throughout the adapter's lifetime.

Recommendation

Remove the `_getDecimals` function and let the constructor revert if `decimals()` fails. Since this is a one-time deployment cost, failing loudly on misconfiguration is preferable to silently assuming a default.

Resolution

`_getDecimals` was removed and the construction calls `baseAggregator.decimals()` directly.

I-3 `isCapped()` conflates not capped with error conditions

Description

The `isCapped()` function returns `false` for three distinct scenarios: no effective snapshot exists, the ratio is within bounds, and `convertToAssets()` reverted:

```
// YieldBearingOracleAdapter.sol:293-301
function isCapped() external view returns (bool) {
    if (!_hasEffectiveSnapshot()) return false;

    try sharesContract.convertToAssets(SHARE_UNIT) returns (uint256
currentRatio) {
        uint256 maxAllowedRatio = _calculateMaxRatioView();
        return currentRatio > maxAllowedRatio;
    } catch {
        return false;
    }
}
```

A caller cannot distinguish "not capped" from "unable to determine."

Recommendation

Document clearly that `false` can indicate either “not capped" or "undetermined."

Resolution

The documentation now clarifies `isCapped` returning false can indicate an undetermined state. `getCurrentSlackBps` was added as an additional function to help distinguish between the two states.

I-4 Reliance on deprecated `answeredInRound` Chainlink field

Description

Both oracle adapters validate `answeredInRound >= roundId` as part of their round validation:

```
// RatioOracleAggregator.sol:212-213
function _ensureComplete(uint256 updatedAt, uint80 roundId, uint80
answeredInRound) internal view {
    if (answeredInRound < roundId) revert CommonChecksLibrary.StaleRound();
```

The `answeredInRound` field is deprecated in the Chainlink API. While the check provides defense-in-depth and the worst case is a false positive (triggering the fallback oracle), relying on deprecated fields creates a maintenance risk if Chainlink changes its behavior.

Recommendation

Remove the `answeredInRound` check in favor of the `updatedAt`-based staleness check which serves the same purpose.

Resolution

`answeredInRound` checks were removed from reverting validation paths. Non-reverting validation paths continue to check it if the value is non-zero as a soft guard.

I-5 Redundant validation checks duplicated across functions

Description

In `YieldBearingOracleAdapter`, the `_validateRound` function checks `updatedAt == 0`, but this is already checked within `_isFresh` (called via `_validateFreshness`), which checks `updatedAt == 0 || updatedAt > block.timestamp`:

```
// YieldBearingOracleAdapter.sol:475-481
function _validateRound(int256 answer, uint256 updatedAt, uint80 roundId,
uint80 answeredInRound) internal view {
    if (answer <= 0) revert CommonChecksLibrary.InvalidBasePrice();
    if (answeredInRound < roundId) revert CommonChecksLibrary.StaleRound();
    if (updatedAt == 0) revert CommonChecksLibrary.StaleRound();

    _validateFreshness(updatedAt);
}
```

While the redundancy is harmless, it adds gas cost and obscures which function is the canonical validation authority.

Recommendation

Remove the redundant checks from `_validateRound` and rely on `_validateFreshness` / `_isFresh` for timestamp and freshness validation.

Alternatively, consider adopting `RatioOracleAggregator`'s simpler `_isComplete` / `_ensureComplete` pattern, which consolidates all round validation into a single function. If this logic were moved to `OracleLib`, both adapters could share the same validation code.

Resolution

The redundant checks were removed from `_validateRound`.

I-6 Dead code branch in `_rolloverEffectiveSnapshot`

Description

The third guard in `_rolloverEffectiveSnapshot` checks if the pending snapshot's effective timestamp is before the current active snapshot timestamp:

```
// YieldBearingOracleAdapter.sol:390-394
function _rolloverEffectiveSnapshot() internal {
    if (_pendingSnapshotEffectiveAt == 0) return;
```

```
    if (block.timestamp < _pendingSnapshotEffectiveAt) return;
    if (_pendingSnapshotEffectiveAt < _snapshotTimestamp) return;
```

This condition (`_pendingSnapshotEffectiveAt < _snapshotTimestamp`) should be unreachable. Pending snapshots are always queued with `effectiveAt = block.timestamp + minimumSnapshotDelay`, and `_snapshotTimestamp` is only ever set from a previously pending snapshot's `effectiveAt`. A new pending snapshot is always in the future relative to the active snapshot.

Recommendation

Remove the unnecessary check.

Resolution

The unreachable branch was removed.

I-7 Redundant external calls in `getRoundData` implementations

Description

Both oracle adapters have suboptimal `getRoundData` implementations. `YieldBearingOracleAdapter` makes two calls to the base aggregator: first, `latestRoundData()` to get the latest round ID for validation, then `getRoundData(requestedRoundId)` to fetch the same data again. `RatioOracleAggregator` similarly calls `ratioAggregator.latestRoundData()` for validation, then delegates to `this.latestRoundData()` which internally calls the aggregators a second time.

```
// YieldBearingOracleAdapter.sol:231-237
function getRoundData(uint80 requestedRoundId) external view returns (...)
{
    (uint80 latestRoundId,,,,) = baseAggregator.latestRoundData();
    if (requestedRoundId != latestRoundId) revert
CommonChecksLibrary.StaleRound();
```

```
    (roundId, answer, startedAt, updatedAt, answeredInRound) =
baseAggregator.getRoundData(requestedRoundId);
    _validateRound(answer, updatedAt, roundId, answeredInRound);
    answer = _computeSharesPrice(answer);
}
```

Since `requestedRoundId` must equal `latestRoundId` (or the function reverts), the second call is redundant. The data can be fetched once and the round ID validated afterward before returning.

Recommendation

Delegate to `this.latestRoundData()` first, then validate that the returned `roundId` matches `requestedRoundId`. This eliminates redundant external calls in both implementations.

Resolution

`getRoundData` in both adapters delegates to `this.latestRoundData`, then validates the round ID, eliminating the redundant calls.

I-8 Magic number for maximum decimals should be a shared constant

Description

`YieldBearingOracleAdapter` defines `MAX_DECIMALS` as a private constant, but `RatioOracleAggregator` uses the raw literal 18. Since both contracts share `OracleLib` for other constants, the maximum decimals constant should live there as well.

Recommendation

Add `MAX_DECIMALS = 18` and any other relevant constants to `OracleLib` so they can be referenced from both contracts.

Resolution

`MAX_DECIMALS`, `MIN_DECIMALS`, and `DECIMAL_BASE` were added to OracleLib and are referenced from there by both adapters.

I-9 Parameter order in `_ensureComplete` does not match `latestRoundData` return order

Description

The `_ensureComplete` function in `RatioOracleAggregator` takes parameters in the order `(updatedAt, roundId, answeredInRound)`:

```
// RatioOracleAggregator.sol:211-212
function _ensureComplete(uint256 updatedAt, uint80 roundId, uint80
answeredInRound) internal view {
```

This does not match the Chainlink `latestRoundData()` return order of `(roundId, answer, startedAt, updatedAt, answeredInRound)`. Call sites must reorder the arguments, which is error-prone.

Recommendation

Reorder parameters to match the Chainlink return convention: `(roundId, updatedAt, answeredInRound)`.

Resolution

`_ensureComplete` and `_isComplete` now take `(roundId, updatedAt)` as parameters, matching the Chainlink order while also dropping `answeredInRound` entirely.

I-10 Confusing snapshot state terminology and misleading function names

Description

The snapshot system uses inconsistent terminology for its states. The code has two explicit storage states (pending via `_pendingSnapshotEffectiveAt != 0`, and active via `_snapshotRatio > 0`), but the logic introduces an implicit third state "effective" (a pending snapshot whose time has arrived but hasn't been committed to storage yet):

```
// YieldBearingOracleAdapter.sol:437-442
function _hasEffectiveSnapshot() internal view returns (bool) {
    if (_maxAnnualGrowthBps == 0) return false;
    if (_snapshotRatio > 0) return true;
    return _isPendingEffective();
}
```

Additionally, `_calculateMaxRatioView` is named as a `view` variant, implying there is a state-mutating counterpart. In reality, the closest mutating equivalent is `_rolloverEffectiveSnapshot`, not a function with a similar name.

Recommendation

Clarify the state model with explicit documentation. Consider renaming `_calculateMaxRatioView` to `_calculateMaxRatio` since there is no non-view variant.

Resolution

The function was renamed to _calculateMaxRatio. Comments were added to give additional context on the snapshot lifecycle.

I-11 Reverting function used in view path where non-reverting variant exists

Description

`_calculateMaxRatioView` calls `_computeGrowthRate` when a pending snapshot is effective:

```
// YieldBearingOracleAdapter.sol:420-423
if (_isPendingEffective()) {
    snapshotRatio = _pendingSnapshotRatio;
    snapshotTimestamp = _pendingSnapshotEffectiveAt;
    growthRate = _computeGrowthRate(snapshotRatio, _maxAnnualGrowthBps);
}
```

The contract provides `_tryComputeGrowthRate` which returns `(0, false)` on overflow instead of reverting. While there is no actual exploit path here, as `updateSnapshot()` validates the growth rate at queue time, and `setGrowthCapBounds()` re-validates pending snapshots against new bounds, using the reverting variant in a view function called from `_computeSharesPriceSafe` is inconsistent with the defensive error-handling pattern used elsewhere.

The same pattern appears in `_rolloverEffectiveSnapshot`.

Recommendation

Remove `_tryComputeGrowthRate` entirely if the reverting variant is considered always safe given the validation coverage. Alternatively, for consistency with the defensive coding style used in `_computeSharesPriceSafe`, consider using `_tryComputeGrowthRate` in `_calculateMaxRatioView` and falling back to the active snapshot's growth rate if computation fails.

Resolution

`_calculateMaxRatio` was updated to use the non-reverting `_tryComputeGrowthRate` function.