

October 29th, 2025

Echelon Market

Comprehensive Security Assessment

Table of Contents

Disclaimer

1. Introduction

- 1.1. Executive Summary
- 1.2. Project Timeline
- 1.3. Scope
- 1.4. Overview of Findings

2. Findings

- 2.1. [H-1] Reserve fee rounding mints extra SY
- 2.2. [M-1] Discount factor can exceed 100%, causing oracle aborts
- 2.3. [M-2] Circuit breaker ignores SY crashes because of stale PY index
- 2.4. [M-3] Missing Time-Based PY Index Caching Enables PT/SY Market Manipulation
- 2.5. [L-1] Approximation rejects valid single-point search window
- 2.6. [L-2] State Mutation inside #[view] Previews in Token Hub
- 2.7. [I-1] Duplicated input validation in calc_slope
- 2.8. [I-2] get_exchange_rate PT-for-SY guard allows equality
- 2.9. [I-3] Mislabelled PT Pause Error In Token Hub
- 2.10. [I-4] Remove fee metadata entry after withdrawal
- 2.11. [I-5] Linear Discount Oracle DoS via Zero TWAP Lookback

Disclaimer

This security assessment represents a time-boxed security review using tooling and manual review methodologies. Our findings reflect our comprehensive evaluation of the materials provided in-scope and are specific to the commit hashes referenced in this report.

The scope of this security assessment is strictly limited to the code explicitly specified in the report. External dependencies, integrated third-party services, libraries, and any other code components not explicitly listed in the scope have not been reviewed and are excluded from this assessment.

Any modifications to the reviewed codebase, including but not limited to smart contract upgrades, protocol changes, or external dependency updates will require a new security assessment, as they may introduce considerations not covered in the current review.

In no event shall Plainshift's aggregate liability for all claims, whether in contract or any other theory of liability, exceed the Services Fee paid for this assessment. The client agrees to hold Plainshift harmless against any and all claims for loss, liability, damages, judgments and/or civil charges arising out of exploitation of security vulnerabilities in the contracts reviewed.

By accepting this report, you acknowledge that deployment and implementation decisions rest solely with the client. Any reliance upon the information in this report is at your own discretion and risk. This disclaimer is governed by and construed in accordance with the laws specified in the engagement agreement between Plainshift and the client.

1. Introduction

Plainshift is a full-stack security firm built on the “shift left” security philosophy. We often work with teams early in the product development process to bring security to a greater organizational range than just smart contracts. From the web app, to fuzzing/formal verification, to a team’s operational security, full-stack security can only be achieved by first understanding there is no “scope” to fully protect the users that trust you.

We’re here to meaningfully revolutionize how teams approach security and guide them towards a holistic approach rather than the single sided approach so prevalent today.

Learn more about us at <https://plainshift.io>.

1.1. Executive Summary

Plainshift was tasked with reviewing the Echelon codebase, with emphasis on Echelon yield trading module, math module, and oracle module. The engagement ran from October 6th 2025 to October 29th 2025. The assessment produced 1 high, 3 medium, and 2 low severity issues, along with 5 informational notes.

Alongside our traditional audit process, we compared Pendle Finance’s reference implementation against Echelon’s implementation and documented any deviations with potential impact. The most severe and confusing class of issues involved arithmetic rounding behavior. Besides the high severity rounding bug we found, Echelon also intentionally applies different rounding directions in several calculations to favor protocol outcomes for security, which leads to discrepancies but not security bugs. The overall quality of the codebase is great.

1.2. Project Timeline

Date	Phase
October 6th, 2025	Kickoff
October 29th, 2025	Audit End
October 29th, 2025	Delivery

1.3. Scope

Repositories	https://github.com/EchelonMarket/echelon-modules
Version	be7f85f07cf108b99b2efe2bf270e83d78231fc8
Contracts	/tree/main/echelon_yield_trading /tree/main/echelon_yield_trading_math /tree/wesley/tha-2806-pendle-draft-pt-pricing-implementation/echelon_yield_trading_oracle
Type	Move
Platform	Aptos

1.4. Overview of Findings

Our comprehensive review yielded 1 high, 3 medium, 2 low severity issues alongside 5 informational findings.

Severity/Impact Level	Count
● High	1
● Medium	3
● Low	2
● Informational	5



2. Findings

2.1. [H-1] Reserve fee rounding mints extra SY

Target	<code>echelon_yield_trading_math/sources/market_math.move</code>
Severity	● High
Category	Rounding issue, math lib integration issue

2.1.1. Description

`market_math::calc_trade()` rounds the reserve-fee share with `fixed_point64::decode()`, which rounds to the nearest integer. When the reserve receives 100% of the fee and the fractional fee value lies in `(0.5, 1)`, the reserve is minted a full extra SY even though the trade only generated a fractional amount of value. The pool effectively gives away more SY than the trade removes, so a trader can drain the pool for free by repeating these small swaps.

The problematic line is `let net_asset_to_reserve = fixed_point64::decode(...)` in `echelon_yield_trading_math/sources/market_math.move`. `decode` is a round-to-nearest helper. When `fee_rate = 1.25`, `reserve_fee_percent = 1.0`, `pt_supply = total_asset = 10_000`, and the trader sells `net_pt_to_account = 11`, the pre-fee asset output is ~ 2.74 and the fee is ~ 0.55 . Because `reserve_fee_percent = 1.0`, the reserve share is equal to the fee (0.55). `decode()` rounds this up to `1`, so the function emits `(net_sy_to_account=2, net_sy_fee=0, net_sy_to_reserve=1)`. The total SY minted is `3`, which exceeds the pre-fee asset amount (~ 2.74). The new regression test `calc_trade_reserve_fee_rounding_mints_extra_sy` added to `echelon_yield_trading_math/tests/market_math_tests.move` asserts both the rounding mismatch and the over-minting using the same curve math as the production code. The test currently passes, confirming the buggy behavior.

2.1.2. Impact

An attacker can loop precise `exact_pt_for_sy` trades that create fractional fees just above 0.5 SY. Each iteration mints one whole extra SY for the reserve while only removing ~ 0.55 SY of value from the pool, steadily depleting pool reserves. The attack requires no upfront capital beyond gas and can drain all SY liquidity.

2.1.3. Recommendation

Use `fixed_point64::decode_round_down` when converting the reserve share to integer SY, or otherwise accumulate the fractional remainder so that the reserve never receives more SY than the fee generated. This prevents over-minting and restores the invariant that user + fee emissions never exceed the real-valued pre-fee asset output.

2.1.4. Remediations

Fixed.

2.1.5. Appendix: PoC

Add the following code to `echelon_yield_trading_math/tests/market_math_tests.move` :

```

1  #[test]
2  fun calc_trade_reserve_fee_rounding_mints_extra_sy() {
3      let py_index = fixed_point64::to_u128(fixed_point64::one());
4      let net_pt_to_account = 11;
5      let pt_supply = 10000;
6      let total_asset = 10000;
7      let rate_scalar = fixed_point64::to_u128(fixed_point64::one());
8      let rate_anchor_fp = fixed_point64::encode(4);
9      let rate_anchor = i128::from(fixed_point64::to_u128(rate_anchor_fp));
10     let fee_rate = fixed_point64::to_u128(fixed_point64::fraction(5, 4));
11     let reserve_fee_percent = fixed_point64::to_u128(fixed_point64::one());
12
13     let (net_sy_to_account, net_sy_fee, net_sy_to_reserve) = market_math::calc_trade(
14         py_index,
15         net_pt_to_account,
16         pt_supply,
17         total_asset,
18         rate_scalar,
19         rate_anchor,
20         fee_rate,
21         reserve_fee_percent,
22         true,
23     );
24
25     // Reconstruct the fractional fee and reserve share using the same curve maths
26     let numerator = pt_supply + net_pt_to_account;
27     let proportion = fixed_point64::div(fixed_point64::encode(numerator), pt_supply +
28         total_asset);
29     let logit_p = fixed_point64::div_fp(proportion,
30         fixed_point64::sub_fp(fixed_point64::one(), proportion));
31     let (sign, ln_proportion) = log_exp_math::ln(logit_p);
32     let price_curve_basis = fixed_point64::div_fp(ln_proportion,
33         fixed_point64::from_u128(rate_scalar));
34     let exchange_rate_i128 = if (sign == 1) {
35         i128::add(i128::from(fixed_point64::to_u128(price_curve_basis)), rate_anchor)
36     } else {
37         i128::sub(rate_anchor, i128::from(fixed_point64::to_u128(price_curve_basis)))
38     };
39     let pre_fee_exchange_rate_fp =
40         fixed_point64::from_u128(i128::abs_u128(exchange_rate_i128));
41 }

```

```
38     let pre_fee_asset_to_account =
39         fixed_point64::div_fp(fixed_point64::encode(net_pt_to_account),
40         pre_fee_exchange_rate_fp);
41     let fee = fixed_point64::mul_div_fp(
42         pre_fee_asset_to_account,
43         fixed_point64::sub_fp(fixed_point64::from_u128(fee_rate),
44         fixed_point64::one()),
45         fixed_point64::from_u128(fee_rate),
46     );
47     let reserve_share = fixed_point64::mul_fp(fee,
48     fixed_point64::from_u128(reserve_fee_percent));
49
50     let minted_total = net_sy_to_account + net_sy_fee + net_sy_to_reserve;
51     let minted_total_fp = fixed_point64::encode(minted_total);
52     let reserve_share_rounded = fixed_point64::encode(net_sy_to_reserve);
53
54     assert!(net_sy_to_account == 2, 0);
55     assert!(net_sy_fee == 0, 0);
56     assert!(net_sy_to_reserve == 1, 0);
57     assert!(fixed_point64::lt(&reserve_share, &reserve_share_rounded), 0);
58     assert!(fixed_point64::gt(&minted_total_fp, &pre_fee_asset_to_account), 0);
59 }
```

Run PoC: `aptos move test --package-dir echelon_yield_trading_math`

2.2. [M-1] Discount factor can exceed 100%, causing oracle aborts

Target echelon_yield_trading_oracle/sources/oracle.move

Severity ● Medium

Category Vulnerability

2.2.1. Description

Echelon yield trading oracle's `set_discount_rate_per_year` ignores `time-to-maturity`, causing potential DoS and diverges from the AAVE Capo base implementation. The root cause is that `linear_discount_oracle::current_discount` multiplies the annual rate by time-to-maturity without bounding the product. Markets with >1 year to expiry and perfectly valid discount rates will trigger an abort in `get_price_unsafe`, rendering the oracle unusable.

`current_discount` computes discount to be `time_to_maturity*discount_rate_per_year/IMPLIED_RATE_TIME`. `set_discount_rate_per_year` only enforces `discount_rate_per_year` to be not greater than 1:

```
1  assert!(
2      fixed_point64::lte(
3          &fixed_point64::from_u128(discount_rate_per_year),
4          &fixed_point64::one()
5      ),
6      ERR_ORACLE_INVALID_DOWNSCALING_FACTOR
7  );
```

For a market expiring in 18 months and `discount_rate_per_year = 0.8`, for example, `current_discount` returns 1.2. `fixed_point64::sub_fp(1, 1.2)` aborts, so every price query fails.

The AAVE Capo implementation that Echelon based on explicitly checks time-to-maturity, blocking any configuration that would produce a $\geq 100\%$ discount. The Move port omitted this guard, so configurations accepted by the Solidity contract (rate ≤ 1) can still brick the oracle if maturity is far enough out.

```
1 // PendlePriceCapAdapter::_setDiscountRatePerYear
2 if (
3     ((MATURITY - block.timestamp) * discountRatePerYear_) / SECONDS_PER_YEAR
4     >= PERCENTAGE_FACTOR
5 ) {
6     revert DiscountExceeds100Percent();
7 }
```

`_setDiscountRatePerYear` is called in constructor, making the configuration denied from the start. Echelon should enforce a similar mechanism.

2.2.2. Impact

- Denial of service for price reads on long-dated markets, even with reasonable rates.
- Downstream systems relying on `get_price` (risk engine, lending caps) fail closed.
- No runtime recovery without code changes; every call to the oracle aborts until the discount is reconfigured to something artificially small.

2.2.3. Recommendation

Add the maturity-aware cap. Compute `time_to_maturity` wherever `discount_rate_per_year` is set (both initializer and setter). Abort if `(time_to_maturity * discount_rate_per_year) / IMPLIED_RATE_TIME >= FixedPoint64::one()`.

2.2.4. Remediations

Fixed.

2.3. [M-2] Circuit breaker ignores SY crashes because of stale PY index

Target	<code>echelon_yield_trading_oracle/sources/oracle.move</code>
Severity	● Medium
Category	Vulnerability

2.3.1. Description

The linear discount oracle’s PT concentration circuit breaker is designed to trip when PT dominates liquidity or when PT liquidity vanishes. The breaker uses `market_pt_proportion`, which converts SY into PT-equivalent value with `token_hub::py_index_current`. That helper locks the PY index to the highest historical rate (`max(sy_exchange_rate, stored_py_index)`). When the SY redemption rate falls sharply (e.g., rebasing token de-pegs), the breaker still evaluates using the stale, high PY index, so the PT proportion stays moderate and the breaker never fires.

The breaker reads PT share via `market_pt_proportion` (`echelon/echelon_yield_trading/sources/market.move:1469`). That routine converts SY into “PT-equivalent” units using the current PY index and then computes

```
pt_ratio = PT_balance / (PT_balance + SY_balance * PY_index)
```

`token_hub::py_index_current` (`echelon/echelon_yield_trading/sources/token_hub.move:1285`) does not follow the live SY redemption rate on the way down. It stores

```
PY_index = max(SY_redemption_rate_now, PY_index_cached)
```

Once the SY rate spikes upward, this max clamps the index at the peak value, even if SY later de-pegs.

When SY crashes, `SY_balance * PY_index` still uses the old, large index. The denominator stays big, so `pt_ratio` hardly moves and never crosses the breaker thresholds (`oracle.move:330-335`).

2.3.2. Impact

- Lenders relying on the breaker to halt price queries during an SY collapse get a stale, over-optimistic PT valuation instead of the expected circuit-break status.

- This undermines the risk envelope promised by the breaker: downstream protocols keep accepting inflated PT collateral and are exposed to under-collateralisation.

2.3.3. Recommendation

Keep the monotonic PY index for interest accounting, but let the breaker logic reference the live SY redemption rate (e.g. compute PT share with `SY_balance * current_SY_rate` instead of the cached peak).

2.3.4. Remediations

Fixed.

2.4. [M-3] Missing Time-Based PY Index Caching Enables PT/SY Market Manipulation

Target	echelon_yield_trading_oracle/sources/token_hub.move
Severity	● Medium
Category	Fairness

2.4.1. Description

Echelon's `py_index_current()` function queries the Thala oracle on every call without caching, unlike Pendle V2's `doCacheIndexSameBlock` mechanism. When an external oracle update transaction executes within the same second as user transactions, subsequent calls to `py_index_current()` receive updated values while earlier calls in the same second used stale values. Since `py_index` directly affects PT/SY AMM exchange rates through the `total_asset` calculation, this creates exploitable arbitrage opportunities.

The Thala oracle price doesn't change spontaneously - it requires explicit update transactions from oracle relayers (for Pyth/Switchboard) or authorized updaters (for Simple Oracle). However, sophisticated MEV actors can monitor pending oracle updates and sandwich their trades around these updates, exploiting the PT/SY rate differential to extract value from liquidity providers and other traders.

2.4.1.1. Oracle Price Flow

The system fetches oracle prices through the following call chain:

```

1 token_hub::py_index_current()
2   -> token_hub::sy_exchange_rate()
3     -> oracle::get_and_update_price_by_name() // Thala oracle @ 0x092e95...
4       -> tiered_oracle::get_and_update_price()
5         -> [Tier 1] Pyth/Switchboard/Chainlink (primary, ~1-5 second updates)
6         -> [Tier 2] Simple Oracle (fallback, minimum 30 second interval)

```

Oracle name format: "RR:<underlying_address>.<base_address>"

2.4.1.2. Current Implementation Without Caching

Echelon (token_hub.move:1268-1278):

```

1 public fun py_index_current(token_set_obj: Object<TokenSet>): u128 {
2     let yield_token_interest = borrow_global_mut<YieldTokenInterest>(token_set_address);
3
4     // Queries oracle every call - no caching
5     let py_index = math128::max(sy_exchange_rate(token_set.sy_obj),
6     yield_token_interest.py_index);
7     yield_token_interest.py_index = py_index;
8
9     py_index
10 }
11
12 public fun sy_exchange_rate(sy_obj: Object<StandardYieldToken>): u128 {
13     // Calls external Thala oracle - can return different values per call
14     fixed_point64::to_u128(
15         oracle::get_and_update_price_by_name(
16             names::rr_oracle_name(underlying_metadata, base_metadata)
17         )
18     )
19 }

```

Pendle V2 (PendleYieldToken.sol:389-399):

```

1 function _pyIndexCurrent() internal returns (uint256 currentIndex) {
2     // Block-level caching with optional flag
3     if (doCacheIndexSameBlock && pyIndexLastUpdatedBlock == block.number)
4         return _pyIndexStored;
5
6     uint128 index128 = PMath.max(IStandardizedYield(SY).exchangeRate(),
7     _pyIndexStored).Uint128();
8     _pyIndexStored = index128;
9     pyIndexLastUpdatedBlock = uint128(block.number);
10
11     return index128;
12 }

```

2.4.1.3. How py_index Affects PT/SY Market Pricing

The PT/SY AMM uses `py_index` to convert SY reserves to asset terms for pricing
(`market_math.move:get_market_pre_compute`):

```
1 public fun get_market_pre_compute(py_index: u128, pt_supply: u64, sy_supply: u64, ...):  
  (...) {  
2     let total_asset = pt_math::sy_to_asset(py_index, sy_supply);  
3     let rate_anchor = get_rate_anchor(pt_supply, ..., total_asset, ...);  
4     // ...  
5 }
```

When `py_index` changes, `total_asset` changes proportionally, affecting the PT/SY exchange rate in the AMM. This creates an arbitrage opportunity when oracle updates occur within the same block or second.

2.4.1.4. Attack Scenario

Attacker monitors mempool for oracle update transactions and sandwiches trades around the update:

```
1 Second T (timestamp = 1000):  
2 TX 1 (Attacker Front-run): Swap 1,000 PT -> 950 SY at py_index=1.0  
3 TX 2 (Oracle Relay): Pyth/Switchboard update pushes new price (py_index -> 1.1)  
4 TX 3 (Attacker Back-run): Swap 950 SY -> 1,050 PT at py_index=1.1  
5 Net profit: 50 PT from rate differential
```

All three transactions execute within the same second (timestamp 1000), but due to lack of caching:

- TX 1 queries oracle, gets `py_index` = 1.0 (before update)
- TX 2 updates on-chain oracle state
- TX 3 queries oracle, gets `py_index` = 1.1 (after update)

The attacker exploits the change in `total_asset` calculation between transactions, profiting from the PT/SY exchange rate shift. This requires:

1. Monitoring pending oracle update transactions
2. Front-running with a trade at the old price
3. Back-running with a reverse trade at the new price

2.4.2. Impact

1. **MEV Extraction:** Sophisticated actors can arbitrage PT/SY market when oracle updates within same block/second, extracting value from liquidity providers and traders.

2. **Fairness Violations:** Users executing identical operations (minting PT/YT, redeeming interest) within same time unit receive different amounts based on oracle timing.
3. **Unpredictable Pricing:** Transaction outcomes cannot be reliably previewed if oracle updates between simulation and execution.

Attack Feasibility:

- **Prerequisites:**
 - MEV infrastructure to monitor mempool for oracle update transactions
 - Sufficient PT/SY pool liquidity to execute meaningful trades
 - Oracle relayer must submit update transaction (external, not attacker-controlled)
- **Likelihood:** Low-Medium
 - Depends on oracle update frequency (Tier 1: ~1-5 seconds, Tier 2: 30s minimum)
 - Requires precise timing - attacker must sandwich their trades around oracle update
 - Multiple transactions must fit within same second (Aptos timestamp granularity)
 - On Aptos, blocks are produced every ~0.5-1 seconds, so multiple blocks can share same `timestamp::now_seconds()` value
- **Impact:** Medium (extractable value proportional to rate change and liquidity depth)

Mitigating Factors:

- Oracle price doesn't change spontaneously - requires explicit update transaction from relayer
- Users cannot directly manipulate oracle (external Thala/Pyth/Switchboard feeds)
- Within-transaction consistency maintained via `max()` operation
- Attack window is narrow - must execute within same second as oracle update
- Aptos's fast block times (~0.5-1s) mean fewer transactions per second compared to slower chains

2.4.3. Recommendation

Implement time-based caching for `py_index` to ensure consistent rates within the same time unit. Since Aptos uses `timestamp::now_seconds()` with second-level granularity, and oracle updates require explicit transactions, per-second caching ensures all transactions within the same second see the same `py_index` value regardless of when oracle updates execute.


```
1 struct YieldTokenInterest has key {
2     // ... existing fields ...
3     py_index_last_updated_timestamp: u64,
4 }
5
6 public fun py_index_current(token_set_obj: Object<TokenSet>): u128 {
7     let yield_token_interest = borrow_global_mut<YieldTokenInterest>(token_set_address);
8
9     // Return cached value if already updated this second
10    if (yield_token_interest.py_index_last_updated_timestamp == timestamp::now_seconds()) {
11        return yield_token_interest.py_index
12    };
13
14    // Query oracle and update cache
15    let py_index = math128::max(sy_exchange_rate(token_set.sy_obj),
16    yield_token_interest.py_index);
17    yield_token_interest.py_index = py_index;
18    yield_token_interest.py_index_last_updated_timestamp = timestamp::now_seconds();
19
20    py_index
21 }
```

This ensures all transactions within the same second use the same `py_index`, preventing arbitrage opportunities and fairness violations.

2.4.4. Remediations

Fixed.

2.5. [L-1] Approximation rejects valid single-point search window

Target	echelon_yield_trading_math/sources/market_approximation_math.move
--------	---

Severity	● Low
----------	-------

Category	Vulnerability
----------	---------------

2.5.1. Description

In `market_approximation_math::get_first_guess/scale_clamp`, the binary search aborts if the search window is a single point (`start = end`).

```
1  assert!(guess_min < guess_max,  
    ERR_MARKET_APPROXIMATION_INVALID_APPROX_PARAMS_MIN_GTE_MAX);
```

In Pendle, `validateApprox` only enforces `guessMin <= guessMax` and the binary search will evaluate the lone candidate (`calcMidpoint` handles `min == max`). Here, legitimate trades near the boundary where `guess_min == guess_max` (for example when `sy_math::sy_to_asset(py_index, sy_in_amount)` exactly equals the market's `calc_max_pt_in`) abort immediately, even though a solution exists.

2.5.2. Impact

Users can't execute SY→YT swaps in these edge states; practically this is a denial-of-service for a slice of valid swaps. However, the edge case is rare and the impact is low.

2.5.3. Recommendation

Relax the guard to permit equality (mirror Pendle's `<=` check) and let the loop evaluate the single guess, or explicitly short-circuit when `guess_min == guess_max` by returning that value.

2.6. [L-2] State Mutation inside #[view] Previews in Token Hub

Target echelon_yield_trading/sources/token_hub.move

Severity ● Low

Category Offchain Usability

2.6.1. Description

In `token_hub.move`, some preview functions flagged as read-only will call helpers that mutate on-chain state.

```

1  #[view]
2  public fun preview_redeem_py_token(...) acquires ... {
3      // ... existing code ...
4      // Update post-exiry data. If market expired, factor into redeemable calculation
5      update_post_expiry_data(token_set_obj);
6      // ... existing code ...
7  }
8
9  #[view]
10 public fun preview_redeem_due_interest(...) acquires ... {
11     // ... existing code ...
12     distribute_interest(primary_fungible_store::primary_store(user_addr, yt_metadata),
13                         user_addr, yt_metadata(token_set_obj), index);
13     // ... existing code ...
14 }
```

Both `update_post_expiry_data` and `distribute_interest` write to `yield_token_interest`, which is globally borrowed mutably via `borrow_global_mut<YieldTokenInterest>(token_set_address)`.

2.6.2. Impact

Moderate usability risk. Any client relying on read-only RPCs receives aborted transactions instead of preview data, preventing UX features like estimating redemption outcomes.

2.6.3. Recommendation

Split the mutating logic into separate, non-view helpers and keep the previews purely read-only.

2.6.4. Remediations

Fixed.

2.7. [I-1] Duplicated input validation in `calc_slope`

Target	<code>echelon_yield_trading_math/sources/market_approximation_math.move</code>
Severity	● Informational
Category	Code Quality

2.7.1. Description

In `market_approximation_math::calc_slope`, there are duplicated input validations:

```

1  public fun calc_slope(total_asset: u64, pt_supply: u64, pt_to_market: u64, rate_scalar:
    u128, rate_anchor: I128): I128 {
2      assert!(total_asset > pt_to_market, ERR_MARKET_APPROXIMATION_INVALID_PT_TO_MARKET);
3      assert!(rate_scalar > 0, ERR_MARKET_APPROXIMATION_INVALID_RATE_SCALAR);
4
5      let diff_asset_pt_to_market = total_asset - pt_to_market;
6      let sum_pt = pt_supply + pt_to_market;
7
8      assert!(diff_asset_pt_to_market > 0 && sum_pt > 0, ERR_INVALID_PT_TO_MARKET);

```

The first assertion on `total_asset > pt_to_market` ensures that `diff_asset_pt_to_market` is positive, making the subsequent check on `diff_asset_pt_to_market > 0` redundant.

2.7.2. Recommendation

Remove `diff_asset_pt_to_market > 0` in the last assertion to eliminate redundancy.

2.7.3. Remediations

Fixed.

2.8. [I-2] `get_exchange_rate` PT-for-SY guard allows equality

Target `echelon_yield_trading_math/sources/market_math.move`

Severity ● Informational

Category Vulnerability

2.8.1. Description

The `get_exchange_rate` function in `market_math.move` allows `net_pt_to_account` equal the full PT supply. Under equality, the guard passes, but `log_proportion` later receives a zero numerator and aborts with an unrelated error.

```

1  if (!exact_pt_for_sy) {
2      assert!(pt_supply >= net_pt_to_account, ERR_MARKET_MATH_NET_PT_TO_ACCOUNT_TOO_LARGE);
3  };
4
5  let numerator = if (exact_pt_for_sy) pt_supply + net_pt_to_account else pt_supply -
    net_pt_to_account;
6
7  let proportion = fixed_point64::div(fixed_point64::encode(numerator), pt_supply +
    total_asset);
8
9  assert!(fixed_point64::lte(&proportion, &fixed_point64::from_u128(MAX_MARKET_PROPORTION)),
    ERR_MARKET_MATH_PT_PROPORTION_TOO_LARGE);
10
11  // @audit: log(0) aborts with math error
12  let (sign, ln_proportion) = log_proportion(fixed_point64::to_u128(proportion));

```

Pendle implementation avoids this issue:

```

1  function executeTradeCore(...) {
2      // ...
3      if (market.totalPt <= netPtToAccount)
4          revert Errors.MarketInsufficientPtForTrade(market.totalPt, netPtToAccount);

```

2.8.2. Recommendation

Change the guard to `assert!(pt_supply > net_pt_to_account, ...)` and add a regression test that proves the custom error triggers. This improves the debuggability and UI experience of the protocol.

2.8.3. Remediations

Fixed.

2.9. [I-3] Mislabeled PT Pause Error In Token Hub

Target	echelon_yield_trading/sources/token_hub.move
--------	--

Severity	● Informational
----------	-----------------

Category	Vulnerability
----------	---------------

2.9.1. Description

Both PT transfer hooks guard on the PT pause flag but surface `ERR_TOKEN_HUB_YT_TRANSFER_PAUSED`, producing misleading incident logs even though behavior is correct.

```
1 public fun deposit_pt<T: key>(...) acquires EchelonTokenHub {
2     assert(!pt_transfer_paused(), ERR_TOKEN_HUB_YT_TRANSFER_PAUSED);
3     // ...existing code...
4 }
5
6 public fun withdraw_pt<T: key>(...) acquires EchelonTokenHub {
7     assert(!pt_transfer_paused(), ERR_TOKEN_HUB_YT_TRANSFER_PAUSED);
8     // ...existing code...
9 }
```

In `token_hub.move`, both functions call `pt_transfer_paused()` but abort with `ERR_TOKEN_HUB_YT_TRANSFER_PAUSED`, so responders see a YT-specific error for a PT-only pause.

2.9.2. Recommendation

Add a new error code `ERR_TOKEN_HUB_PT_TRANSFER_PAUSED` and use it in the PT transfer hooks to improve clarity in error logs.

2.9.3. Remediations

Fixed.

2.10. [I-4] Remove fee metadata entry after withdrawal

Target echelon_yield_trading/sources/market.move

Severity ● Informational

Category Vulnerability

2.10.1. Description

`collect_fee` records each fee asset in `EchelonYieldMarkets.fees_metadata` but nothing prunes the list, so it accumulates every asset that ever accrued fees. Although only managers can append to the list, this is a valid maintenance concern since the vector can grow unbounded.

```

1  fun collect_fee(fee: FungibleAsset) acquires EchelonYieldMarkets {
2      // ...
3      if (!vector::contains(&echelon_yield_markets.fees_metadata, &metadata)) {
4          vector::push_back(&mut echelon_yield_markets.fees_metadata, metadata);
5      };
6      primary_fungible_store::deposit(package::package_address(), fee);
7  }
8
9  public fun withdraw_fee(manager: &signer, asset_metadata: Object<Metadata>): FungibleAsset
   acquires EchelonYieldMarkets {
10     assert!(manager::is_manager(manager), ERR_MARKET_UNAUTHORIZED);
11
12     assert!(vector::contains(&borrow_mut_echelon_yield_markets().fees_metadata,
13                             &asset_metadata), ERR_MARKET_WITHDRAW_FEES_INVALID_ASSET);
14
15     let amount = primary_fungible_store::balance(package::package_address(),
16                                                  asset_metadata);
17     primary_fungible_store::withdraw(&package::get_signer(), asset_metadata, amount)
18     // @audit: should remove asset_metadata from fees_metadata if balance is 0
19 }

```

2.10.2. Recommendation

After a successful withdrawal, remove that entry from `fees_metadata` .

2.10.3. Remediations

Fixed.

2.11. [I-5] Linear Discount Oracle DoS via Zero TWAP Lookback

Target	echelon_yield_trading_oracle/sources/oracle.move
--------	--

Severity	● Informational
----------	-----------------

Category	Admin error
----------	-------------

2.11.1. Description

The linear discount oracle lets managers set the TWAP lookback window to zero. When this happens, every downstream price query aborts because the periphery oracle rejects a zero-second lookback. A single misconfiguration therefore bricks PT pricing across the system.

- `set_twap_lookback_seconds` in `echelon/echelon_yield_trading_oracle/sources/oracle.move` accepts any `twap_lookback_seconds` value without validation.
- `get_price` / `get_price_unsafe` later call `oracle_periphery::consult_oracle`, which asserts `seconds_ago > 0` (see `echelon/echelon_yield_trading/sources/oracle_periphery.move:60`).
- If a manager sets the lookback to `0`, every subsequent price request (including health checks) aborts with `ERR_ORACLE_ZERO_SECONDS_AGO`, effectively freezing the oracle.
- Pendle's analogue continues to function with zero duration by falling back to the cached implied rate; Echelon's port needs explicit guarding instead.

2.11.2. Impact

Any manager or compromised governance action that sets the lookback to zero renders PT price feeds unusable. Integrations that depend on `get_price` revert, preventing lending liquidations, circuit breakers, and valuation logic from executing.

2.11.3. Recommendation

Reject non-positive lookback windows when updating the configuration, e.g. add `assert!(twap_lookback_seconds > 0, ERR_ORACLE_ZERO_SECONDS_AGO)` inside `set_twap_lookback_seconds`. Optionally cap the window to a sane upper bound to match Pendle's expectations.

2.11.4. Remediations

Fixed.