SPEARBIT

Pendle-Finance Security Review

Auditors

Hyh, Lead Security Researcher

Kurt Barry, Lead Security Researcher

Xiaoming90, Security Researcher

Mario Poneder, Associate Security Researcher

Report prepared by: Lucas Goiriz

July 26, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Pendle is a permissionless yield-trading protocol where users can execute various yield-management strategies.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of pendle-core-v2 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 15 days in total, Pendle Finance engaged with Spearbit to review the pendle-core-v2 protocol. In this period of time a total of **12** issues were found.

Summary

Project Name	Pendle Finance
Repository	pendle-core-v2
Commit	4d83f4...b264
Type of Project	Yield, DeFi
Audit Timeline	May 15 to Jun 6
Two week fix period	Jul 3

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	2	2	0
Low Risk	5	3	2
Gas Optimizations	1	0	1
Informational	3	0	3
Total	**12**	**6**	**6**

5 Findings

5.1 High Risk

5.1.1 Expanding reward token list right after linked YT expiration freezes the accumulated rewards

Severity: High Risk

Context: RewardManagerAbstract.sol#L44-L65

Description: In some cases it is allowed to expand SY reward tokens list and this can happen after `_setPostExpiryData()` has fixed the expiry state in one of the YTs linked to that SY.

For example, if a new reward token was added to the Nitro pool, but not yet added to `rewardTokens` of CamelotV1Volatile SY, attacker can run any YT operation involving `_setPostExpiryData()` in the first block with its `expiry <= block.timestamp` and then run public `updateRewardTokensList()` of SY:

- PendleCamelotV1VolatileSY.sol#L91-L104:

```
 function _getRewardTokens() internal view virtual override returns (address[] memory res) { //
↪   <<
     return rewardTokens;
 }

 /// @notice allows anyone to add new rewardTokens to this SY if a new rewardToken is added to
↪   the Nitro pool
 function updateRewardTokensList() public virtual {
     if (nitroPool == address(0)) return; // if nitroPool is not set, we don't need to update
↪   rewardTokens list

     address token1 = ICamelotNitroPool(nitroPool).rewardsToken1().token;
     address token2 = ICamelotNitroPool(nitroPool).rewardsToken2().token;

     if (token1 != address(0) && token1 != xGRAIL && !rewardTokens.contains(token1))
↪   rewardTokens.push(token1); // <<
     if (token2 != address(0) && token2 != xGRAIL && !rewardTokens.contains(token2))
↪   rewardTokens.push(token2); // <<
 }
```

After that reward redeeming will be blocked for all the holders of this YT as any `redeemDueInterestAndRewards() -> _updateAndDistributeRewards()` call will revert:

- PendleYieldToken.sol#L160-L169:

```
 function redeemDueInterestAndRewards(
     address user,
     bool redeemInterest,
     bool redeemRewards
 ) external nonReentrant updateData returns (uint256 interestOut, uint256[] memory rewardsOut) {
     if (!redeemInterest && !redeemRewards) revert Errors.YCNothingToRedeem();

     // if redeemRewards == true, this line must be here for obvious reason
     // if redeemInterest == true, this line must be here because of the reason above
     _updateAndDistributeRewards(user); // <<
```

- RewardManagerAbstract.sol#L35-L41

```
function _updateAndDistributeRewardsForTwo(address user1, address user2) internal virtual {
    (address[] memory tokens, uint256[] memory indexes) = _updateRewardIndex(); // <<
    if (tokens.length == 0) return;

    if (user1 != address(0) && user1 != address(this)) _distributeRewardsPrivate(user1, tokens,
↪  indexes); // <<
    if (user2 != address(0) && user2 != address(this)) _distributeRewardsPrivate(user2, tokens,
↪  indexes);
}
```

- PendleYieldToken.sol#L472-L480:

```
function _updateRewardIndex() internal override returns (address[] memory tokens, uint256[]
↪  memory indexes) {
    tokens = getRewardTokens();
    if (isExpired()) {
        indexes = new uint256[](tokens.length);
        for (uint256 i = 0; i < tokens.length; i++) indexes[i] =
↪  postExpiry.firstRewardIndex[tokens[i]]; // <<
    } else {
        indexes = IStandardizedYield(SY).rewardIndexesCurrent();
    }
}
```

As `index = postExpiry.firstRewardIndex[new_token] == 0`, not being initialized, while `userIndex` will be set to `INITIAL_REWARD_INDEX.Uint128() == 1`, and `deltaIndex = index - userIndex = 0 - 1`:

- RewardManagerAbstract.sol#L44-L65:

```
function _distributeRewardsPrivate(address user, address[] memory tokens, uint256[] memory
↪  indexes) private {
    assert(user != address(0) && user != address(this));

    uint256 userShares = _rewardSharesUser(user);

    for (uint256 i = 0; i < tokens.length; ++i) {
        address token = tokens[i];
        uint256 index = indexes[i]; // <<
        uint256 userIndex = userReward[token][user].index;

        if (userIndex == 0) {
            userIndex = INITIAL_REWARD_INDEX.Uint128(); // <<
        }

        if (userIndex == index) continue;

        uint256 deltaIndex = index - userIndex; // <<
        uint256 rewardDelta = userShares.mulDown(deltaIndex);
        uint256 rewardAccrued = userReward[token][user].accrued + rewardDelta;

        userReward[token][user] = UserReward({index: index.Uint128(), accrued:
↪  rewardAccrued.Uint128()});
    }
```

- RewardManagerAbstract.sol#L16:

```
uint256 internal constant INITIAL_REWARD_INDEX = 1;
```

Impact: since `rewardTokens` list is append only and `_setPostExpiryData()` can't be run again, all the rewards within `userRewardOwed` balances will be permanently frozen in the YT contract.

Likelihood: Low (SY with an expanding reward token list and not yet added reward token is a prerequisite) + Impact: Critical (most of the rewards are end up frozen) = Severity: High.

Recommendation: Consider ignoring not initialized indices, e.g.:

- RewardManagerAbstract.sol#L44-L65:

```
    function _distributeRewardsPrivate(address user, address[] memory tokens, uint256[] memory
↪   indexes) private {
        assert(user != address(0) && user != address(this));

        uint256 userShares = _rewardSharesUser(user);

        for (uint256 i = 0; i < tokens.length; ++i) {
            address token = tokens[i];
            uint256 index = indexes[i];
            uint256 userIndex = userReward[token][user].index;

            if (userIndex == 0) {
                userIndex = INITIAL_REWARD_INDEX.Uint128();
            }

  -         if (userIndex == index) continue;
  +         if (userIndex == index || index == 0) continue;

            uint256 deltaIndex = index - userIndex;
            uint256 rewardDelta = userShares.mulDown(deltaIndex);
            uint256 rewardAccrued = userReward[token][user].accrued + rewardDelta;

            userReward[token][user] = UserReward({index: index.Uint128(), accrued:
↪   rewardAccrued.Uint128()});
        }
```

Pendle: Resolved in commit 4e6983ab of PR 526.

Spearbit: Verified.

IMPORTANT NOTE:

- *The issue only affects SY that can add rewards tokens. None of the active markets are affected by this issue.*
- *Principal of PT remains withdrawable even if the issue occurs.*

5.2 Medium Risk

5.2.1 User can be denied interest income due to interest amount rounding

Severity: Medium Risk

Context: InterestManagerYT.sol#L75-L79

Description: `redeemDueInterestAndRewards()` can be triggered not in the best interests of a user. If the exchange rate is big enough in L2 environment this can form a griefing surface, given that there is a precision reduction with rounding down in interest calculation (and the minimal increment figure, `currentIndex - prevIndex`, can have much less magnitude than `prevIndex` and `currentIndex`):

- `_distributeInterestPrivate()`, InterestManagerYT.sol#L75-L79:

```
  uint256 principal = _YTbalance(user);

  uint256 interestFromYT = (principal * (currentIndex - prevIndex)).divDown(prevIndex *
↪   currentIndex);

  userInterest[user].accrued += interestFromYT.Uint128();
```

- PMath.sol#L48-L53:

```
function divDown(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 aInflated = a * ONE;
    unchecked {
        return aInflated / b;
    }
}
```

For example, an attacker can call `redeemDueInterestAndRewards(user, true, false)` for a target `user` after every known IBT index update (i.e. after every `IStandardizedYield(SY).exchangeRate()` uptick), minimizing the interest for them via rounding.

In a somewhat stretched, yet theoretically possible, example, suppose Bob the user has YT balance of `38e12` (in 18 dp, so it's `38e-6` of the one whole unit), with underlying being some very old ETH liquid staked derivative with `prevIndex = 9e6 * 1e18` (say ETH worth `10000 USD` and one whole unit of this derivative was worth `9e6 * 10000 = 90 bln USD`, while Bob's position was worth `38e12 * 9e6 * 10000 / 1e18 = 3_420_000 USD`), with current rate and index being `exchangeRate() = currentIndex = 9e6 * e18 + 2.08e18`, where `9e24 * 0.01 / (60 * 24 * 30) = 2.08e18` (rounded for brevity). That corresponds to once per hour updates yielding `1 p.p.` in one month. Let's suppose that this is close to typical increment of this index, given the increment frequency and its median yield (so it's `1%` monthly, close to `12%` APY).

Then `interestFromYT = 38e12 * 2.08e18 * 1e18 / ((9e24 + 2.08e18) * 9e24) = 0` (`0.976...` truncated). If attacker keeps this up for a month, calling `redeemDueInterestAndRewards()` once an hour, then `0.01 * 38e12 * 9e6 * 10000 / 1e18 = 0.01 * 3420000 = 34200 USD` worth of interest was stolen from Bob.

In order for attacker to spend less than Bob loses, so the griefing be viable, it should cost less than `34200 / (60 * 24 * 30) = 0.79 USD` to perform each call, which is 10-50 times higher than current L2 costs.

The key assumption here is the usage of this inflated index, but apart from some derivative being very old there might be other more immediate reasons for such a grouping (e.g. one unit of the index can start with some minimal stake or with some initial capitalization, and so on, as there are no obligations for such derivatives to be tied to one monetary unit).

Impact: any user with balance of eligible magnitude can be denied all the interest income as a griefing. This income will not be retrievable as only explicitly defined after expiry income, `totalSyInterestForTreasury`, is due for the treasury, all the other excess income, including rounding residual, which is amplified by the attack, is frozen with the contract.

Likelihood: Low (the key prerequisite is index being substantially inflated) + Impact: High (the whole interest income of a target user can be frozen) = Severity: Medium.

Recommendation: Consider restricting `redeemDueInterestAndRewards()` calls to the beneficiaries and also forbidding the zero `interestFromYT` case.

Pendle: Resolved in commit 6144fb49 and 855602bf of PR 526.

Spearbit: Verified.

5.2.2 REDACTED

The issue was identified by Spearbit, resolved by Pendle, and verified as resolved by Spearbit.

5.3 Low Risk

5.3.1 Rounding down of the amounts used in liquidity provision logic allows for stealing from market rare side when gas is cheap enough

Severity: Low Risk

Context: MarketMathCore.sol#L144-L156, MarketMathCore.sol#L158

Description: Truncation of the `syUsed/ptUsed` variables can allow to receive LP without supplying a rare side of the market. Supplying low enough `ptDesired` (with big `syDesired` that will be discarded) to have `syUsed == 0` or low enough `syUsed` to have `ptUsed == 0` can be repeated many times in L2 environment, while burning can be done at bulk thereafter, stealing the aggregated position of this rare side from the existing LPs.

The high proportion (rare asset) case is guarded by controlling for `MAX_MARKET_PROPORTION = (1e18 * 96) / 100`. Using it as a simple example and ignoring fees, say `totalLp = 100e18`, `totalPt = 95e18`, `totalSy = 5e18` is a current non-manipulated market state (PT is cheap and equilibrium interest rate is high). Bob the attacker can run `mint() -> addLiquidity()` with `ptDesired = 18 (18 wei)`, `syDesired = 1e36`, having `netLpByPt = (ptDesired * market.totalLp) / market.totalPt = 18 * 100e18 / 95e18 = 18`, `netLpByPt < netLpBySy`, `lpToAccount = ptUsed = 18`, `syUsed = (market.totalSy * lpToAccount) / market.totalLp = (5e18 * 18) / 100e18 = 0`.

Repeating this `1e7` times (there is no impact on the output yet as Bob's share accumulates very slowly, `18 * (100e18 + 18e7) / (95e18 + 18e7) = 18`), Bob spends `18e7` PT and obtains `18e7` LP. They can then `burn()` the whole stake, receiving `netSyToAccount = (lpToRemove * market.totalSy) / market.totalLp = (18e7 * 5e18) / (100e18 + 18e7) = 0.9e7 - 1` SY and `netPtToAccount = (lpToRemove * market.totalPt) / market.totalLp = 18e7 * (95e18 + 18e7) / (100e18 + 18e7) = 17.1e7` PT, with the net impact being spending `18e7 - 17.1e7 = 0.9e7` PT and gaining `0.9e7 - 1` SY. Since PT was cheap this very close to 1-to-1 PT to SY conversion represents a gain for Bob as long as gas costs are low enough.

Likelihood: Medium + Impact: Low = Severity: Low.

Recommendation: Consider rounding up both `syUsed` and `ptUsed` (for example, with `rawDivUp`), and also controlling for the zero side cases, e.g.:

- MarketMathCore.sol#L158:

```
- if (lpToAccount <= 0) revert Errors.MarketZeroAmountsOutput();
+ if (lpToAccount <= 0 || syUsed <= 0 || ptUsed <= 0) revert Errors.MarketZeroAmountsOutput();
```

Pendle: Resolved in commit 95ee8175 and 603b5a5 of PR 526.

Spearbit: Verified.

5.3.2 Asset amount user owes in `swapSyForExactPt` can be understated with rounding down of a positive integer

Severity: Low Risk

Context: MarketMathCore.sol#L258

Description: When `netPtToAccount > 0` rounding down can reduce the asset amount, `preFeeAssetToAccount`, that determines what is owed by the caller.

Recommendation: Consider applying the similar logic as in `netSyToAccount` case, e.g.:

- MarketMathCore.sol#L258

```
- int256 preFeeAssetToAccount = netPtToAccount.divDown(preFeeExchangeRate).neg();
+ int256 preFeeAssetToAccount = netPtToAccount < 0 ?
+                               netPtToAccount.divDown(preFeeExchangeRate).neg() :
+                               (netPtToAccount.Uint() *
↪ PMath.ONE).rawDivUp(preFeeExchangeRate.Uint()).Int().neg();
```

Pendle: Resolved in commit 073a3873 of PR 526.

Spearbit: Verified.

5.3.3 Rebasing down of any reward token can freeze YT and LP transfers, YT and treasury rewards redeeming

Severity: Low Risk

Context: PendleYieldToken.sol#L192-L210, RewardManager.sol#L43

Description:

1. If any reward token is rebasing (e.g. stETH) and gets slashed after `_setPostExpiryData()` execution then there will be no any rewards for treasury from this expired SY until that reward token balance restores as `_selfBalance(tokens[i]) - postExpiry.userRewardOwed[tokens[i]]` will be reverting.

   - PendleYieldToken.sol#L192-L210:

   ```
   function redeemInterestAndRewardsPostExpiryForTreasury()
       external
       nonReentrant
       updateData
       returns (uint256 interestOut, uint256[] memory rewardsOut)
   {
       // ...

       for (uint256 i = 0; i < tokens.length; i++) {
           rewardsOut[i] = _selfBalance(tokens[i]) - postExpiry.userRewardOwed[tokens[i]];
           emit CollectRewardFee(tokens[i], rewardsOut[i]);
       }
   ```

   Since the reward token list is fixed or being append only SY implementations the waiting or manual topping up look to be the only options in that case.

2. If any of the reward tokens be rebased downwards (get slashed), `_updateRewardIndex()` will be similarly blocked until its balance gets restored above `lastBalance`:

- RewardManager.sol#L43:

```
uint256 accrued = _selfBalance(tokens[i]) - lastBalance;
```

This will make unavailable YT and LP token transfers and YT's `redeemDueInterestAndRewards()` via blocking `YT._updateRewardIndex()` and `YT._setPostExpiryData()` as `YT.rewardIndexesCurrent() -> SY.rewardIndexesCurrent() -> RM._updateRewardIndex()` sequence utilized in both cases will revert.

Recommendation: Ensure on each integration that no downside rebasing is possible for reward tokens. In order to guarantee availability of the related functionality consider accommodating for short term balance dips, for example with ignoring downside movements:

1. PendleYieldToken.sol#L192-L210:

```
  function redeemInterestAndRewardsPostExpiryForTreasury()
      external
      nonReentrant
      updateData
      returns (uint256 interestOut, uint256[] memory rewardsOut)
  {
      // ...

      for (uint256 i = 0; i < tokens.length; i++) {
-         rewardsOut[i] = _selfBalance(tokens[i]) - postExpiry.userRewardOwed[tokens[i]];
+         uint256 selfBalance = _selfBalance(tokens[i]);
+         rewardsOut[i] = selfBalance > postExpiry.userRewardOwed[tokens[i]] ? selfBalance -
↪ postExpiry.userRewardOwed[tokens[i]] : 0;
          emit CollectRewardFee(tokens[i], rewardsOut[i]);
  }
```

2. RewardManager.sol#L43:

```
- uint256 accrued = _selfBalance(tokens[i]) - lastBalance;
+ uint256 selfBalance = _selfBalance(tokens[i]);
+ uint256 accrued = selfBalance > lastBalance ? selfBalance - lastBalance : 0;
```

This isn't a complete mitigation as any prolonged downward rebasing in the current implementation will produce an accounting insolvency as late claiming users will not be able to do so. In order to accommodate this case fully dynamic rewards balances are needed, i.e. balances of all the users have to be rebased after total reward balance was (so current fixed accrual based solution needs to take some form of dynamic index-dependent accrual).

Pendle: Acknowledged. The intended design of the reward system is to not support rebasing reward tokens.

Spearbit: Acknowledged.

5.3.4 Interest and reward fee rates are back propagated when changed

Severity: Low Risk

Context: InterestManagerYT.sol#L43-L54, PendleYieldToken.sol#L429-L444

Description: Interest and reward fee rates are applied backwards when changed, i.e. new fee rates are applied to the periods where old fee rates were active:

- InterestManagerYT.sol#L43-L54:

```
function _doTransferOutInterest(
    // ...
) internal returns (uint256 interestAmount) {
    address treasury = IPYieldContractFactory(factory).treasury();
    uint256 feeRate = IPYieldContractFactory(factory).interestFeeRate(); // <<

    // ...

    uint256 feeAmount = interestPreFee.mulDown(feeRate);
```

- PendleYieldToken.sol#L429-L444:

```
function __doTransferOutRewardsLocal(
    // ...
) internal returns (uint256[] memory rewardAmounts) {
    address treasury = IPYieldContractFactory(factory).treasury();
    uint256 feeRate = IPYieldContractFactory(factory).rewardFeeRate(); // <<
    // ...
    for (uint256 i = 0; i < tokens.length; i++) {
        // ...

        uint256 feeAmount = rewardPreFee.mulDown(feeRate);
```

Likelihood: Medium (fees can be changed as a part of usual workflow) + Impact: Medium (fee rates are applied incorrectly) = Severity: Medium.

Recommendation: Consider making the fee rates immutable fields of PendleYieldToken contract `YT` set on construction in `createYieldContract()`.

Pendle: Acknowledged. This was a conscious design choice since we expect to change fees very rarely, if ever. The complexity of not back-propagating the fee outweighed the benefits.

Spearbit: Acknowledged.

5.3.5 LP valuation can be overstated, while `rateOracle` precision can be reduced in `PendleLpOracleLib`

Severity: Low Risk

Context: PendleLpOracleLib.sol#L65-L69, PendleLpOracleLib.sol#L74-L85

Description:

1. `tradeSize` in `_getLpToAssetRateRaw()` can be either positive or negative based on the sign of `cParam.mulDown(comp.totalAsset) - state.totalPt`, where `cParam = LogExpMath.exp(comp.rateScalar.mulDown((rateOracle - comp.rateAnchor))` can vary from `0` to being large enough since its based on the `rateOracle - comp.rateAnchor = rateOracle - newExchangeRate + lnProportion.divDown(rateScalar)`, which can differ depending on the evolution of the rate.

   Positive `tradeSize` means asset was removed from the pool and PT was added. In this case rounding down in `comp.totalAsset - tradeSize.divDown(rateHypTrade)` expression overstates the assets as the removed part is being rounded down, which is the equivalent of the remaining part being rounded up.

2. `rateOracle` is subject to two divisions, reducing its precision, which can be avoided as these operations cancel each other:

   - PendleLpOracleLib.sol#L79:

     ```
         rateOracle = PMath.IONE.divDown(market.getPtToAssetRateRaw(duration).Int());
     ```

   - PendlePtOracleLib.sol#L48:

     ```
         return PMath.ONE.divDown(assetToPtRate);
     ```

Recommendation:

1. Consider conditioning the logic on the direction of the trade, e.g.:

   - PendleLpOracleLib.sol#L65-L69:

     ```
          totalHypotheticalAsset =
              comp.totalAsset -
     -                tradeSize.divDown(rateHypTrade) +
     +                tradeSize > 0 ? (tradeSize.Uint() *
     ↪   PMath.ONE).rawDivUp(rateHypTrade.Uint()).Int() : tradeSize.divDown(rateHypTrade) +
              (state.totalPt + tradeSize).divDown(rateOracle);
     ```

2. Consider streamlining the code, retrieving `rateOracle` directly, e.g.:
    - PendleLpOracleLib.sol#L74-L85:

```
      function _getPtRatesRaw(
          IPMarket market,
          MarketState memory state,
          uint32 duration
      ) private view returns (int256 rateOracle, int256 rateHypTrade) {
    -       rateOracle = PMath.IONE.divDown(market.getPtToAssetRateRaw(duration).Int());
    +       uint256 lnImpliedRate = market.getMarketLnImpliedRate(duration);
    +       uint256 timeToExpiry = expiry - block.timestamp;
    +       rateOracle = MarketMathCore._getExchangeRateFromImpliedRate(lnImpliedRate,
↪   timeToExpiry);
          int256 rateLastTrade = MarketMathCore._getExchangeRateFromImpliedRate(
              state.lastLnImpliedRate,
    -           state.expiry - block.timestamp
    +           timeToExpiry
          );
          rateHypTrade = (rateLastTrade + rateOracle) / 2;
      }
```

    - PendlePtOracleLib.sol#L65:

```
    -  function _getMarketLnImpliedRate(IPMarket market, uint32 duration) private view
↪   returns (uint256) {
    +  function getMarketLnImpliedRate(IPMarket market, uint32 duration) internal view
↪   returns (uint256) {
```

Pendle: Resolved in commit 14c9f26a of PR 526.

Spearbit: Verified. The 2nd recommendation was implemented while the 1st one was intentionally skipped.

5.4 Gas Optimization

5.4.1 Redundant expiry check in `MarketMathCore.sol`

Severity: Gas Optimization

Context: MarketMathCore.sol#L202, MarketMathCore.sol#L224

Description: In `MarketMathCore.sol`, both `executeTradeCore()` and `getMarketPrecompute()` check for expiry of the underlying YT/PT pair. Since `getMarketPrecompute()` is only called by `executeTradeCore()`, these two checks are redundant and one could be removed.

Recommendation: Remove one of the redundant checks (logically, the check in `getMarketPrecompute()` makes the most sense to remove, as that function is intended to compute values needed for the trade computation, not to perform checks).

Pendle: Acknowledged.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 Unreachable instance of MarketExchangeRateBelowOne error

Severity: Informational

Context: MarketMathCore.sol#L314

Description: The following check cannot be triggered since the newExchangeRate cannot be less than PMath.IONE.

- MarketMathCore.sol#L312-L314:

```
int256 newExchangeRate = _getExchangeRateFromImpliedRate(lastLnImpliedRate, timeToExpiry);

if (newExchangeRate < PMath.IONE) revert Errors.MarketExchangeRateBelowOne(newExchangeRate);
```

The newExchangeRate is computed as follows, where always rt >= 0 due to being *unsigned*.

- MarketMathCore.sol#L345-L352:

```
function _getExchangeRateFromImpliedRate(
    uint256 lnImpliedRate,
    uint256 timeToExpiry
) internal pure returns (int256 exchangeRate) {
    uint256 rt = (lnImpliedRate * timeToExpiry) / IMPLIED_RATE_TIME;

    exchangeRate = LogExpMath.exp(rt.Int());
}
```

In the worst case, rt == 0 and therefore exchangeRate == PMath.IONE. Consequently, the above instance of the MarketExchangeRateBelowOne error can never be reached.

Recommendation: Although the above check can be removed, this level of caution might be justified considering potential future changes to the contract. It is suggested to add an inline comment which explains the defensive concerns behind this check.

Pendle: Acknowledged.

Spearbit: Acknowledged.

5.5.2 LP valuation can be understated if YT caches index updates

Severity: Informational

Context: PendleLpOracleLib.sol#L34-L39, PendleLpOracleLib.sol#L52-L54

Description: When expiry <= block.timestamp and syIndex > pyIndex, while YT.doCacheIndexSameBlock() && YT.pyIndexLastUpdatedBlock() == block.number, it will be totalHypotheticalAsset = state.totalPt + state.totalSy * pyIndex / ONE:

- PendleLpOracleLib.sol#L52-L54:

```
if (state.expiry <= block.timestamp) {
    // 1 PT = 1 Asset post-expiry
    totalHypotheticalAsset = state.totalPt + PYIndexLib.syToAsset(PYIndex.wrap(pyIndex),
↪  state.totalSy);
```

- PYIndex.sol#L19-L21:

```
function syToAsset(PYIndex index, uint256 syAmount) internal pure returns (uint256) {
    return SYUtils.syToAsset(PYIndex.unwrap(index), syAmount);
}
```

- SYUtils.sol#L7-L9:

```
function syToAsset(uint256 exchangeRate, uint256 syAmount) internal pure returns (uint256) {
    return (syAmount * exchangeRate) / ONE;
}
```

Then `lpToAssetRateRaw = (state.totalPt + state.totalSy * pyIndex / ONE).divDown(state.totalLp)`:

- PendleLpOracleLib.sol#L34-L39:

```
function getLpToSyRate(IPMarket market, uint32 duration) internal view returns (uint256) {
    (uint256 syIndex, uint256 pyIndex) = PendlePtOracleLib.getSYandPYIndexCurrent(market);
    uint256 lpToAssetRateRaw = _getLpToAssetRateRaw(market, duration, pyIndex);
    if (syIndex >= pyIndex) {
        return lpToAssetRateRaw.divDown(syIndex); // <<
    }
```

and `getLpToSyRate()` returns `(state.totalPt + state.totalSy * pyIndex / ONE).divDown(state.totalLp).divDown(sy`

Simplifying and omitting rounding down, it is `totalPt / syIndex + state.totalSy * pyIndex / syIndex`, i.e. `state.totalSy` is weighted with `pyIndex / syIndex < 1`, underpricing the LP.

Recommendation: Since LP valuation is an additional dependency for index updates consider avoiding enabling `doCacheIndexSameBlock` switch for YTs whose index can have any material movements within the block.

Pendle: Acknowledged.

Spearbit: Acknowledged.

5.5.3 Edge case handling in `OracleLib`

Severity: Informational

Context: OracleLib.sol#L87-L107, OracleLib.sol#L120-L124

Description: In the `OracleLib` library contract, the following instances were uncovered where edge cases of *public* methods are not handled gracefully:

1. In the `binarySearch()` method, which searches for Oracle observations at a given timestamp, there is an infinite loop (`while(true)`) which can lead to a revert (out-of-gas) in case the desired `target` timestamp is newer/older than the stored observations.

2. In the `getSurroundingObservations()` method, which utilizes the aforementioned `binarySearch()` method to get Oracle observations around a given timestamp, there is an edge case where the desired `target` timestamp coincides with the most recent observation (`beforeOrAt.blockTimestamp == target`). However, in this case the method's second return value `atOrAfter` is left unset which could impact future integrations that directly rely on this method.

Recommendation: The following changes are suggested:

1. Declare the `binarySearch()` method as *internal* since it it only used by `getSurroundingObservations()` which handles these edge cases.

2. In case of an exact match, return the same observation twice in OracleLib.sol#L124:

```
- return (beforeOrAt, atOrAfter);
+ return (beforeOrAt, beforeOrAt);
```

Pendle: Acknowledged.

Spearbit: Acknowledged.