[H-0] PendleYieldToken._beforeTokenTransfer() should not call _setPostExpiryData() without checking if isExpired()

https://github.com/pendle-finance/pendle-core-internal-v2/blob/2be5ad0da54f76a637ddd6a223d49aa02cb8c07f/contracts/core/YieldContracts/PendleYieldToken.sol#L405

```
400 function _beforeTokenTransfer(
401         address from,
402         address to,
403         uint256
404     ) internal override {

406         _updateAndDistributeRewardsForTwo(from, to);
407         _distributeInterestForTwo(from, to);
408     }
```

https://github.com/pendle-finance/pendle-core-internal-v2/blob/2be5ad0da54f76a637ddd6a223d49aa02cb8c07f/contracts/core/YieldContracts/PendleYieldToken.sol#L288-L301

```
288 function _setPostExpiryData() internal {
289         PostExpiryData storage local = postExpiry;

291
292         _redeemExternalReward(); // do a final redeem. All the future reward income will belon
293

295         address[] memory rewardTokens = ISuperComposableYield(SCY).getRewardTokens();
296         uint256[] memory rewardIndexes = ISuperComposableYield(SCY).rewardIndexesCurrent();
297         for (uint256 i = 0; i < rewardTokens.length; i++) {
298             local.firstRewardIndex[rewardTokens[i]] = rewardIndexes[i];
299             local.userRewardOwed[rewardTokens[i]] = _selfBalance(rewardTokens[i]);
300         }
301     }
```

By the first time YT token is transferred, `_setPostExpiryData()` will be called and set `firstScyIndex`, without checking if `isExpired()`.

As a result, `postExpiry.firstScyIndex` will be set much earlier, and the value will be lower than expected.

Recommendation

Change `_updateRewardIndex()` to:

https://github.com/pendle-finance/pendle-core-internal-v2/blob/2be5ad0da54f76a637ddd6a223d49aa02cb8c07f/contracts/core/YieldContracts/PendleYieldToken.sol#L384-L397

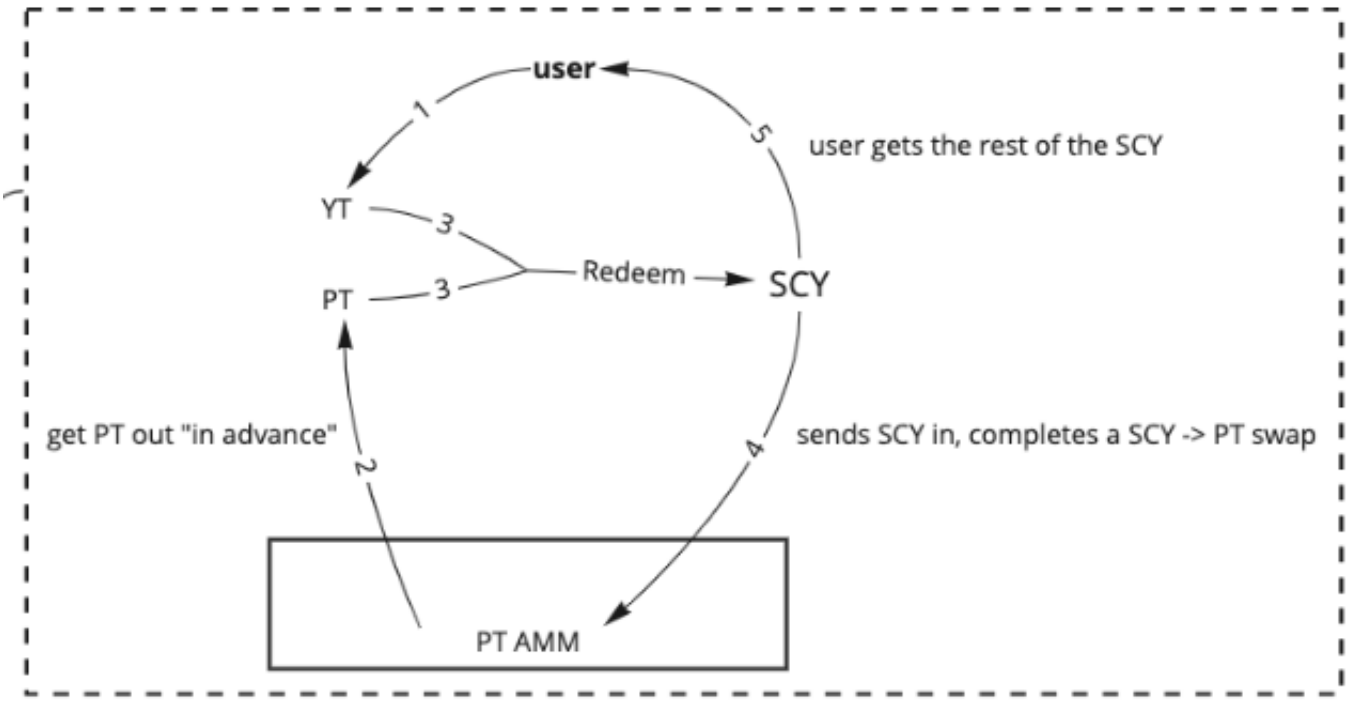
```
384 function _updateRewardIndex()
385         internal
386         override
387         returns (address[] memory tokens, uint256[] memory indexes)
388     {
389         tokens = getRewardTokens();
390         if (isExpired()) {

392             indexes = new uint256[](tokens.length);
393             for (uint256 i = 0; i < tokens.length; i++)
394                 indexes[i] = postExpiry.firstRewardIndex[tokens[i]];
395         } else {
396             indexes = ISuperComposableYield(SCY).rewardIndexesCurrent();
397         }
398     }
```

Since the storage variable `postExpiry` will be read in `_updateRewardIndex()`, it makes more sense to `_setPostExpiryData()` if `isExpired()` in the context of `_updateRewardIndex()` rather than `_setPostExpiryData()` in the context of `_beforeTokenTransfer()`; the gas overhead will be minimal as a hot SLOAD of `postExpiry` only costs 100 gas.

It's also more fit semantically when it's done in `_updateRewardIndex()`, the function name sounds like it may update the reward index.

[M-1] `PendleRouter#_callbackSwapYtForScy` may not pay enough SCY to the PT AMM market

https://pendle.notion.site/How-PendleRouter-works-2c2166bac1784cca81a2c85d796190be



https://github.com/pendle-finance/pendle-core-internal-v2/blob/2be5ad0da54f76a637ddd6a223d49aa02cb8c07f/contracts/core/actions/ActionCallback.sol#L114-L137

```
114 function _callbackSwapYtForScy(
115         address market,
116         int256 ptToAccount,
117         int256 scyToAccount,
118         bytes calldata data
119     ) internal {
120         (address receiver, uint256 minScyOut) = _decodeSwapYtForScy(data);
121         (ISuperComposableYield SCY, , IPYieldToken YT) = IPMarket(market).readTokens();
122
123
124         uint256 scyOwed = scyToAccount.neg().Uint();
125
126         address[] memory receivers = new address[](2);
127         uint256[] memory amountPYToRedeems = new uint256[](2);
128
129
130         (receivers[1], amountPYToRedeems[1]) = (
131             receiver,
132             ptToAccount.Uint() - amountPYToRedeems[0]
133         );
134
135         uint256[] memory amountScyOuts = YT.redeemPYMulti(receivers, amountPYToRedeems);
136         require(amountScyOuts[1] >= minScyOut, "insufficient SCY out");
137     }
```

Expected:

```
market to repay scyOwed SCY;
```

Actual:

$$\frac{scyAmount \cdot exchangeRate_{scy}}{exchangeRate_{yt.scyIndexCurrent()}}$$

When $exchangeRate_{scy} < exchangeRate_{yt.scyIndexCurrent()}$, the transaction will revert at L216-219 due to "insufficient SCY" paid to `market` .

https://github.com/pendle-finance/pendle-core-internal-v2/blob/2be5ad0da54f76a637ddd6a223d49aa02cb8c07f/contracts/libraries/SCY/SCYIndex.sol#L14

```
13 function newIndex(ISuperComposableYield SCY) internal view returns (SCYIndex) {
15     }
```

https://github.com/pendle-finance/pendle-core-internal-v2/blob/2be5ad0da54f76a637ddd6a223d49aa02cb8c07f/contracts/core/YieldContracts/PendleYieldToken.sol#L224-L248

```
224 function _redeemPY(address[] memory receivers, uint256[] memory amountPYToRedeems)
225         internal
226         returns (uint256[] memory amountScyOuts)
227     {
228         uint256 totalAmountPYToRedeem = amountPYToRedeems.sum();
229         IPPrincipalToken(PT).burnByYT(address(this), totalAmountPYToRedeem);
230         if (!isExpired()) _burn(address(this), totalAmountPYToRedeem);
231
233         uint256 totalScyInterestPostExpiry;
234         amountScyOuts = new uint256[](receivers.length);
235
236         for (uint256 i = 0; i < receivers.length; i++) {
237             uint256 scyInterestPostExpiry;
243             totalScyInterestPostExpiry += scyInterestPostExpiry;
244         }
245         if (totalScyInterestPostExpiry != 0) {
246             postExpiry.totalScyInterestForTreasury += totalScyInterestPostExpiry.Uint128();
247         }
248     }
```

https://github.com/pendle-finance/pendle-core-internal-v2/blob/2be5ad0da54f76a637ddd6a223d49aa02cb8c07f/contracts/core/Market/PendleMarket.sol#L193-L222

```
193 function swapScyForExactPt(
```

```
        address receiver,
        uint256 exactPtOut,
        bytes calldata data
    ) external nonReentrant notExpired returns (uint256 netScyIn, uint256 netScyToReserve) {
        MarketState memory market = readState(true);

        (netScyIn, netScyToReserve) = market.swapScyForExactPt(
            SCY.newIndex(),
            exactPtOut,
            block.timestamp
        );

        IERC20(PT).safeTransfer(receiver, exactPtOut);
        IERC20(SCY).safeTransfer(market.treasury, netScyToReserve);

        _writeState(market);

        if (data.length > 0) {
            IPMarketSwapCallback(msg.sender).swapCallback(exactPtOut.Int(), netScyIn.neg(), da
        }

        // have received enough SCY




        emit Swap(receiver, exactPtOut.Int(), netScyIn.neg(), netScyToReserve);
    }
```

Recommendation

Change `ActionCallback._callbackSwapYtForScy()` L122 to: $exchangeRate_{yt.scyIndexCurrent()}$

[I-2] Manipulatable `SCY.exchangeRate()` can be dangerous for PY tokens

https://github.com/pendle-finance/pendle-core-internal-v2/blob/2be5ad0da54f76a637ddd6a223d49aa02cb8c07f/contracts/core/YieldContracts/PendleYieldToken.sol#L185-L189

```
/// @dev maximize the current rate with the previous rate to guarantee non-decreasing rate
function scyIndexCurrent() public returns (uint256 currentIndex) {
    currentIndex = Math.max(ISuperComposableYield(SCY).exchangeRate(), _scyIndexStored);
    _scyIndexStored = currentIndex.Uint128();
}
```

The design of non-decreasing rate with `scyIndexCurrent()` works pretty well with normal SCYs, even if they have some drawdown, the PY tokens and PT AMM market can still continue to work quite fairly.

However, we find it can be quite dangerous if the SCY's `exchangeRate()` is manipulatable.

For example, if the SCY's exchangeRate is reading the current `balanceOf` underlying token as part of the `totalAsset` or `totalValue` div by the `totalSupply` or `totalShares`, and somehow the `mint()` function allows external calls after the tokens has been transferred in and before the new shares are minted. (Usually enabled by features like swap mint, hookable mint, flash mint, etc.)

In the external call, the attacker can call `PendleYieldToken.sol#scyIndexCurrent()` to update the `_scyIndexStored` to the temporary inflated exchange rate.

And the underlying tokens transferred in to infalte the temporary exchangeRate can be clawback by redeeming the newly minted shares right after.

This would allow the attacker to inflate the `exchangeRate` of the SCY at 0 cost and set the `scyIndexCurrent()` to an unrealistic high value.