SHERLOCK

Security Review For Own Protocol

Collaborative Audit Prepared For: **Own Protocol**
Lead Security Expert(s): **0xeix**
hildingr
pkqs90
Date Audited: **November 3 - November 13, 2025**

Introduction

Own is a permissionless DeFi protocol that issues ERC-20 tokens representing synthetic exposure to real-world equities. Users can mint these tokens by depositing yield-bearing stablecoins, receiving price exposure to the underlying asset while paying a floating interest rate.

The protocol functions as a two-sided marketplace: users obtain equity exposure on-chain, and liquidity providers supply the backing capital and earn the interest paid by users. Pool solvency and backing are maintained through an LP-driven rebalancing mechanism that ensures each synthetic asset remains fully collateralized relative to its reference price.

Scope

Repository: own-protocol/own-contracts

Audited Commit: e4a8f6bd24595b3ff96bdf8dfeeb791445c814ad

Final Commit: d65486abd8fd2e79d85409ec71d9c04662ce65b4

Files:

- src/protocol/AssetOracle.sol
- src/protocol/AssetPoolFactory.sol
- src/protocol/AssetPool.sol
- src/protocol/PoolCycleManager.sol
- src/protocol/PoolLiquidityManager.sol
- src/protocol/PoolStorage.sol
- src/protocol/ProtocolRegistry.sol
- src/protocol/strategies/DefaultPoolStrategy.sol
- src/protocol/xToken.sol

Final Commit Hash

d65486abd8fd2e79d85409ec71d9c04662ce65b4

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
8	6	12

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: LP can withdraw collateral immediately after submitting liquidity request, creating unbacked liquidity [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/10

Vulnerability Detail

When an LP calls `addLiquidity()`, they submit an `ADD_LIQUIDITY` request which deposits the required collateral. However, the `liquidityCommitment` is not updated until the request is resolved during the next rebalance cycle via `resolveRequest()`.

In `reduceCollateral()`, the function calculates the required collateral based on the current `liquidityCommitment`:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L282

```
function reduceCollateral(uint256 amount) external nonReentrant onlyRegisteredLP {
    LPPosition storage position = lpPositions[msg.sender];
    uint256 lpCollateral = position.collateralAmount;
    if (amount == 0 || amount > lpCollateral) revert InvalidWithdrawalAmount();

    uint256 requiredCollateral =
    ↪  poolStrategy.calculateLPRequiredCollateral(address(this), msg.sender);
    if (lpCollateral - amount < requiredCollateral) {
        revert InsufficientCollateral();
    }
    ...
}
```

The issue is that `calculateLPRequiredCollateral()` uses `lpPositions[lp].liquidityCommitment` which is still zero for pending requests:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/strategies/DefaultPoolStrategy.sol#L377

```
function calculateLPRequiredCollateral(address liquidityManager, address lp)
↪  external view returns (uint256) {
    IPoolLiquidityManager manager = IPoolLiquidityManager(liquidityManager);
    uint256 lpCommitment = manager.getLPLiquidityCommitment(lp);
    uint256 healthyCollateral = Math.mulDiv(lpCommitment, lpHealthyCollateralRatio,
    ↪  BPS);
    return healthyCollateral;
}
```

Then there's an attack vector to create liquidity out of thin air: in an `cycle == ACTIVE` state, LP first calls `addLiquidity()` and submits a `RequestType.ADD_LIQUIDITY` request, then immediately calls `reduceCollateral()` to withdraw collateral. Because the `lpPositions[lp].liquidityCommitment` is still zero (request not resolved yet), the `requiredCollateral` would also be zero.

Impact

LP can create liquidity commitments without providing collateral, effectively getting free liquidity exposure

Recommendation

Also check for pending liquidity when reducing collateral.

Issue H-2: PoolLiquidityManager yield calculations are incorrect when deducting collateral from LP. [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/11

Vulnerability Detail

During settlement rebalance, collateral may be deducted from LP's balance. This is effectively withdrawing collateral for an LP. When yield-bearing reserves are enabled, the yield corresponding to the deducted collateral should be given back to LP, but this logic currently doesn't exist.

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L415

```
function deductFromCollateral(address lp, uint256 amount) external
↪   onlyPoolCycleManager {
    if (!isLP[lp]) revert NotRegisteredLP();

    LPPosition storage position = lpPositions[lp];
    if (position.collateralAmount < amount) revert InvalidAmount();

    position.collateralAmount -= amount;
    totalLPCollateral -= amount;
    aggregatePoolReserves -= amount;

    reserveToken.safeTransfer(address(assetPool), amount);
}
```

Impact

LP lose yield when collateral is deducted during settlement rebalance.

Recommendation

Calculate the amount of yield accrued by the reduced amount of collateral, and give back to LP.

Issue H-3: PoolLiquidityManager's reserve yield index calculation is incorrect, causing LPs to lose earned yield [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/12

Vulnerability Detail

The protocol uses `reserveYieldIndex` to track accumulated yield from yield-bearing reserves (e.g., aUSDC). This index is updated in `_updateReserveYieldIndex()` to account for yield accrued between updates:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L807

```
function _updateReserveYieldIndex() internal {
    uint256 reserveBalanceBefore = reserveToken.balanceOf(address(this));
    uint256 yIndex = poolStrategy.calculateYieldAccrued(
        aggregatePoolReserves,
        reserveBalanceBefore,
        aggregatePoolReserves
    );
    aggregatePoolReserves = reserveBalanceBefore;
    reserveYieldIndex += yIndex;
}
```

The `calculateYieldAccrued()` function calculates: `(currentAmount - prevAmount) / depositAmount`

The critical flaw is using `aggregatePoolReserves` as the denominator, which changes with every update. This breaks the yield compounding calculation. An example is:

1. Initial state: 100 tokens deposited, `reserveYieldIndex = 1e18`
2. Balance grows to 150: `yIndex = (150-100)/100 = 0.5e18`, `reserveYieldIndex = 1.5e18`
3. Balance grows to 200: `yIndex = (200-150)/150 = 0.333e18`, `reserveYieldIndex = 1.833e18`
4. An LP who deposited 100 at start receives: `100 * 1.833 = 183.3 tokens`
5. Correct amount should be: `100 * (200/100) = 200 tokens`

The denominator should remain constant (like `totalLPCollateral`) to properly track the multiplicative growth of deposits.

Impact

LP loses yield.

Recommendation

Use `totalLPCollateral` as the denominator and don't sync it with reserveToken balance.

Issue H-4: Inactive LPs can be rebalanced to prematurely trigger new cycle, preventing active LPs from rebalancing [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/13

Vulnerability Detail

The protocol tracks the number of active LPs at the start of each rebalancing period via `cycleLPCount`. A new cycle begins when `rebalancedLPs == cycleLPCount`, indicating all active LPs have been rebalanced.

When an LP removes all their liquidity, `isLPActive[lp]` is set to `false` and `lpCount` is decremented. However, `isLP[lp]` remains `true` to preserve historical records.

In both `rebalanceLP()` and `forceRebalanceLP()`, the validation only checks `isLP[lp]` instead of `isLPActive[lp]`:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L346
https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L431

```
    function rebalanceLP(address lp) external {
        ...
@>      if (!poolLiquidityManager.isLP(lp)) revert NotLP();
        ...
    }

    function forceRebalanceLP(address lp) external {
        ...
@>      if (!poolLiquidityManager.isLP(lp)) revert NotLP();
        ...
    }
```

However, we should be using `poolLiquidityManager.isActiveLP(lp)` here for check. The current implementation allows malicious users to rebalance an inactive LP take up a cycleLPCount slot, and the PoolCycleManager will think we have finished rebalancing with all active LPs and start a new cycle.

Impact

A cycle can start with active LPs being unrebalanced, and requests in PoolLiquidityManager unresolved, breaking the entire accounting of the protocol.

Recommendation

Change the check to `isActiveLP(lp)` during rebalance.

Issue H-5: Token split adjustments use wrong user address when operations are performed on behalf of others [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/14

Vulnerability Detail

When a token split occurs, `_splitCheck()` adjusts user positions based on the split multiplier:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetPool.sol#L699

```
function _splitCheck(UserPosition storage position) internal {
    position.assetAmount = poolStrategy.calculatePostSplitAmount(address(this),
    ↪  msg.sender, position.assetAmount);
    userSplitIndex[msg.sender] = poolCycleManager.poolSplitIndex();
}
```

The function always uses `msg.sender` for calculations. However, three functions allow operations on behalf of other users while calling `_splitCheck()`:

1. `liquidationRequest(address user, ...)` - Anyone can liquidate another user's position
2. `claimAsset(address user)` - Anyone can claim assets for another user
3. `claimReserve(address user)` - Anyone can claim reserves for another user

When Bob liquidates Alice's position, `_splitCheck()` uses Bob's `userSplitIndex` to calculate Alice's `assetAmount` and updates Bob's index instead of Alice's. This leaves Alice's position permanently desynchronized after splits.

Impact

User positions become incorrect after token splits when operations are performed on their behalf.

Recommendation

Use the correct user's splitIndex.

Issue H-6: Funds can be locked in the contract when halted [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/29

Summary

User with collateral than required will have funds permantley locked when the protocol is halted.

Vulnerability Detail

When the protocol enters a halted state it is not possible to add any collateral

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/a8e10d9294125053e2b9537baa63fb17f570c466/own-contracts/src/protocol/AssetPool.sol#L155

The `claimAsset()` call will revert if collateral is less than required

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/a8e10d9294125053e2b9537baa63fb17f570c466/own-contracts/src/protocol/AssetPool.sol#L401

But the `exitPool()` call to remove funds when the pool is halted requires no pending requests

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/a8e10d9294125053e2b9537baa63fb17f570c466/own-contracts/src/protocol/AssetPool.sol#L497

Impact

Users' funds are locked because they can not clear the request if they don't have enough collateral when the pool enters a halted state

Tool Used

Manual Review

Recommendation

Allow the ability to add collateral when pool is halted such that user's can process the request and exit the pool

Issue H-7: Token splits between request and claim cause incorrect redemption amounts [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/38

Vulnerability Detail

When users submit redemption or liquidation requests, the request is recorded with the current cycle. Later, when claiming reserves via `claimReserve()`, the protocol uses the rebalance price from the original request cycle:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetPool.sol#L440

```
function claimReserve(address user) external nonReentrant {
    uint256 requestCycle = request.requestCycle;
    // ...
@>  uint256 rebalancePrice = poolCycleManager.cycleRebalancePrice(requestCycle);

    amount = poolStrategy.calculatePostSplitAmount(address(this), user, amount);
    RedemptionValues memory r = _calculateRedemptionValues(user, amount,
    ↪  rebalancePrice, requestCycle);
}
```

If a token split occurs between request submission and claim, the `amount` is correctly adjusted via `calculatePostSplitAmount()`, but the `rebalancePrice` remains from the pre-split cycle. This causes a mismatch in value calculations. An example is:

1. Cycle 5: User submits redemption for 100 tokens at price $10 = $1,000 value
2. Cycle 10: 2:1 token split occurs (100 tokens → 200 tokens, price $10 → $5)
3. Cycle 15: User claims - `amount` adjusted to 200, but `rebalancePrice` still $10
4. Calculated value: 200 × $10 = $2,000 instead of correct 200 × $5 = $1,000

Impact

Users claiming after token splits will end up with wrong amount of reserve tokens.

Recommendation

Store the split index at request time and use the rebalance price from the cycle after any splits occurred, or adjust the rebalance price proportionally based on split ratios.

Discussion

pkqs90

Per protocol team, this issue will be acknowledged, and the token split feature will NOT be used in v1.

Issue H-8: Cumulative interest index calculation doesn't account for token splits causing incorrect interest debt [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/39

Vulnerability Detail

The protocol tracks interest debt using `cumulativeInterestIndex`, which represents the amount of reserve tokens owed per asset token. When a new cycle ends, the index is incremented based on the cycle's interest:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L596

```
if (assetToken.totalSupply() > 0) {
@>  cumulativeInterestIndex[cycleIndex] += Math.mulDiv(cycleInterestAmount *
↪  reserveToAssetDecimalFactor, price, assetToken.totalSupply());
}
```

User interest debt is calculated as: `assetAmount * (poolIndex - userIndex)`. The calculation assumes a linear relationship between token amounts and index values.

However, when token splits occur, `assetToken.totalSupply()` changes but existing `cumulativeInterestIndex` values don't adjust. After a 10:1 split:

- User's `assetAmount` increases 10x (e.g., 100 → 1,000)
- Old index delta remains unchanged (e.g., 0.2e18)
- Interest debt incorrectly increases 10x: `1000 * 0.2e18 = 200` instead of `20`

The same issue affects reserve yield index calculations for yield-bearing tokens.

Impact

Users' interest debt is calculated incorrectly after token splits, with debt scaling proportionally to the split ratio. A 10:1 split causes 10x debt increase, potentially making all positions instantly liquidatable and causing protocol insolvency.

Recommendation

Non-trivial to fix. One idea is to track split-adjusted indices separately to maintain the correct debt-to-token relationship.

Discussion

pkqs90

Per protocol team, this issue will be acknowledged, and the token split feature will NOT be used in v1.

Issue M-1: Delegates cannot rebalance on behalf of LPs due to incorrect modifier check [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/15

Vulnerability Detail

LPs can set delegates who are authorized to call `rebalancePool()` on their behalf:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L237

```
function rebalancePool(address lp, uint256 rebalancePrice) public onlyActiveLP {
    address delegate = poolLiquidityManager.lpDelegates(lp);
@>  if (lp != msg.sender && (delegate == address(0) || msg.sender != delegate))
↪   revert UnauthorizedCaller();
    // ...
}
```

The function correctly allows `msg.sender` to be either the LP or their delegate. However, the `onlyActiveLP` modifier checks if `msg.sender` is an active LP:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L179

```
modifier onlyActiveLP() {
@>  if (!poolLiquidityManager.isLPActive(msg.sender)) revert NotLP();
    _;
}
```

When a delegate calls the function, the modifier checks if the **delegate** is an active LP, not the actual LP whose position is being rebalanced. This causes the function to revert even when the delegate is properly authorized, since delegates are typically not LPs themselves.

Impact

The delegate feature is completely broken - delegates cannot perform their intended role of rebalancing on behalf of LPs. LPs who set delegates expecting automated rebalancing will miss rebalancing windows and be forced into settlement rebalancing with penalties.

Recommendation

Perform the `isLPActive()` check on LP instead on delegate caller.

Issue M-2: LPs receive interest during settlement re-balancing instead of being penalized [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/16

Vulnerability Detail

The protocol has three rebalancing mechanisms with different interest distributions:

1. `rebalancePool()` - LP self-rebalances, receives full interest
2. `rebalanceLP()` - Settlement rebalancing after grace period, should penalize LP
3. `forceRebalanceLP()` - Forced rebalancing after halt threshold, LP loses everything

In `rebalanceLP()`, interest is incorrectly distributed to the LP instead of the protocol:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L389-L391

```
if (lpCycleInterest > 0) {
@>  assetPool.deductInterest(lp, lpCycleInterest, false);
}
```

The third parameter `isSettle` determines interest destination. When `false`, interest goes to the LP. When `true`, it goes to the protocol as penalty. Settlement rebalancing should use `true` to penalize LPs who failed to rebalance during the normal window.

Compare with `forceRebalanceLP()` which correctly uses `true`:

Impact

LPs who miss the rebalancing window are rewarded with full interest instead of being penalized, removing the incentive to rebalance on time.

Recommendation

Change the parameter to `true`.

Issue M-3: Oracle price requests can be spammed to drain Chainlink subscription funds [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/18

Vulnerability Detail

The `requestAssetPrice()` function is publicly callable and uses Chainlink Functions to fetch asset prices. It implements a cooldown mechanism to prevent spam:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetOracle.sol#L94

```
function requestAssetPrice(...) external {
    if (keccak256(abi.encodePacked(source)) != sourceHash) revert InvalidSource();

@>  if (block.timestamp < lastUpdated + REQUEST_COOLDOWN) {
        revert RequestCooldownNotElapsed();
    }
    // ... send request
}
```

The check here uses `lastUpdated`, which is the time when an async request is fulfilled. However, spammers can still spam requests before the request is fulfilled. Example: at T=10, a request is sent, and it returns at T=30, now during 10-30, spammers can spam requests.

Since this function can be called by anyone with a verified subscriptionId, each request will use up LINK funds in the subscription account, and the funds in the account can be drained.

Impact

Attackers can drain all LINK funds from the Chainlink subscription account.

Recommendation

Add access control to this function, since there are offchain keepers maintaining it.

Issue M-4: AssetPool cross-cycle liquidation cancellation causes underflow and incorrect token refunds [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/19

Vulnerability Detail

When a user's position becomes undercollateralized, anyone can submit a liquidation request. Users can cancel pending liquidations by adding collateral to restore their health. The `addCollateral()` function handles this cancellation:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetPool.sol#L173-L184

```
if (collateralHealth == 3) {
    uint256 requestAmount = request.amount;
    address liquidationInitiator = liquidationInitiators[user];
@>  cycleTotalRedemptions -= requestAmount;
    assetToken.safeTransfer(liquidationInitiator, requestAmount);

    delete userRequests[user];
    delete liquidationInitiators[user];
}
```

The function allows cancellation across cycles without checking `request.requestCycle`. However, `cycleTotalRedemptions` is reset to 0 at the end of each cycle in `updateCycleData()`.

Example: In cycle 1, a user was submitted a liquidate request, but didn't call called `claimReserves()` (request is still pending). Then in cycle 2, user tries to add collateral to save his position. But since `cycleTotalRedemptions` is already reset to 0, we will underflow when doing `cycleTotalRedemptions -= requestAmount;`.

Also, if token split happened during the two cycles, we'd be transferring the incorrect amount back to liquidationInitiator.

Impact

Users cannot cancel liquidation requests across cycles due to underflow, and if successful through other means, liquidators receive wrong refund amounts after token splits, breaking protocol accounting.

Recommendation

Only allow liquidation cancellation within the same cycle.

Issue M-5: Inactive LPs cannot re-add liquidity after full withdrawal [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/20

Vulnerability Detail

When an LP first adds liquidity, they are marked as both `isLP[lp] = true` and `isLPActive[lp] = true`, with `lpCount` incremented. If the LP later withdraws all liquidity, `isLPActive[lp]` is set to `false` and `lpCount` is decremented, but `isLP[lp]` remains `true`.

In `addLiquidity()`, the state updates only occur for new LPs:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L172-L181

```
if (isLP[msg.sender]) {
    LPRequest storage request = lpRequests[msg.sender];
    if (request.requestType != RequestType.NONE) revert RequestPending();
} else {
    isLP[msg.sender] = true;
    isLPActive[msg.sender] = true;
    lpCount++;
}
```

When an inactive LP tries to add liquidity again, the first branch executes (since `isLP[msg.sender]` is still `true`), skipping the state updates. The LP's request is created, but they remain inactive and uncounted. During rebalancing, their request will never be resolved since they're not included in `cycleLPCount`.

Impact

LPs who fully withdraw and later re-add liquidity have their funds locked indefinitely as their requests are never resolved.

Recommendation

Set `isLPActive = true` and increment `lpCount` when inactive LPs re-add liquidity:

```
if (isLP[msg.sender]) {
    if (!isLPActive[msg.sender]) {
        isLPActive[msg.sender] = true;
        lpCount++;
    }
```

```
    // ... existing checks
}
```

Issue M-6: The reserve yield is not taking into account when reducing the collateral [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/35

Summary

Currently, there's a check for the `requiredCollateral` that doesn't include the `reserveYield` while the function that determines the health does.

Vulnerability Detail

Let's take a look at the `reduceCollateral()` implementation in the `PoolLiquidityManager`:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L285-282

```
 uint256 lpCollateral = position.collateralAmount;

...

uint256 requiredCollateral =
↪  poolStrategy.calculateLPRequiredCollateral(address(this), msg.sender);
     if (lpCollateral - amount < requiredCollateral) {
         revert InsufficientCollateral();
     }
```

It can be clearly seen that we don't include the `reserveYield` here. So every time when `lpCollateral` (without the `reserveYield` included) falls below the `requiredCollateral`, the liquidity providers won't be able to withdraw the reserve token. In the `getLPCollateralHealth()`, however, the position adds `reserveYield` to the `collateralAmount` and it'll be considered as healthy while at the same time disallowing to withdraw the collateral:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/strategies/DefaultPoolStrategy.sol#L444-461

```
if (isYieldBearing) {
         uint256 reserveYieldIndex = manager.reserveYieldIndex();
         uint256 lpReserveYieldIndex = manager.lpReserveYieldIndex(lp);
         uint256 reserveYieldAmount = Math.mulDiv(
             lpCollateral,
             reserveYieldIndex - lpReserveYieldIndex,
             PRECISION
         );
         lpCollateral += reserveYieldAmount;
     }
```

```
    uint256 healthyLiquidity = Math.mulDiv(lpCommitment, lpHealthyCollateralRatio,
    ↪  BPS);
    uint256 reqLiquidity = Math.mulDiv(lpCommitment, lpLiquidationThreshold, BPS);

    if (lpCollateral >= healthyLiquidity) {
        return 3; // Healthy
    } else if (lpCollateral >= reqLiquidity) {
        return 2; // Warning
```

Here `lpCollateral` is formed as `lpCollateral += reserveYieldAmount`.

Impact

Users can't reduce the collateral when, in fact, the position is considered healthy.

Tool Used

Manual Review

Recommendation

Consider comparing `lpCollateral + reserveYieldAmount` against the `requiredCollateral` or change the implementation of the function that checks the health.

Discussion

bhargavaparoksham

@rodiontr This is definitely not a high vulnerability. It is a medium or a low. Because, the calculateUserRequiredCollateral without including reserveYield expects users / LPs to hold more collateral than needed, which doesn't lead to any serious issues.

So please update the tag accordingly.

rodiontr

> @rodiontr This is definitely not a high vulnerability. It is a medium or a low. Because, the calculateUserRequiredCollateral without including reserveYield expects users / LPs to hold more collateral than needed, which doesn't lead to any serious issues.
> So please update the tag accordingly.

from my perspective, it'll always lead to reverts when reducing the collateral (in the situations where `reserveYield` makes a significant difference) as it's not included but i agree generally it's more of a medium severity

Issue L-1: Collateral calculations round down allowing LPs to underpay required collateral [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/21

Vulnerability Detail

When LPs add liquidity or their required collateral is calculated, the protocol uses default rounding (down) for collateral requirements:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L157

```
uint256 requiredCollateral = Math.mulDiv(amount,
↪  poolStrategy.lpHealthyCollateralRatio(), BPS);
```

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/strategies/DefaultPoolStrategy.sol#L377

```
function calculateLPRequiredCollateral(address liquidityManager, address lp)
↪  external view returns (uint256) {
    uint256 lpCommitment = manager.getLPLiquidityCommitment(lp);
@>  uint256 healthyCollateral = Math.mulDiv(lpCommitment, lpHealthyCollateralRatio,
↪  BPS);
    return healthyCollateral;
}
```

Rounding down benefits the LP by allowing them to provide slightly less collateral than required (by 1 wei).

Impact

No big impact, but it is always preferred to round against the users and for the protocol.

Recommendation

Round up for the above two cases.

Issue L-2: Onchain rebalancing can start before market actually closes due to oracle latency [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/22

Vulnerability Detail

The protocol requires markets to be closed before entering onchain rebalancing. The `initiateOnchainRebalance()` function verifies this using `isMarketOpen()`:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L211

```
function initiateOnchainRebalance() external {
    // ...
@>  if (assetOracle.isMarketOpen()) revert MarketOpen();
}
```

The `isMarketOpen()` check compares the time difference between oracle fulfillment and data timestamp:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetOracle.sol#L203

```
function isMarketOpen() external view returns (bool) {
@>  return (lastUpdated - dataTimestamp) <= MARKET_OPEN_THRESHOLD;
}
```

Where `lastUpdated` is set when the Chainlink Functions callback completes, and `dataTimestamp` comes from the OHLC data itself. If Chainlink Functions experiences delays (network congestion, high gas, etc.), and the request took more than 300 seconds (MARKET_OPEN_THRESHOLD) to process, the market would be assumed as closed, allowing us to enter ONCHAIN_REBALANCE state when the market is actually open.

Impact

Onchain rebalancing can begin while the market is still open, allowing rebalancing with wrong prices.

Recommendation

There's no perfect fix. Need to monitor the response time and update the MARKET_OPEN_THRESHOLD threshold. Ideally make MARKET_OPEN_THRESHOLD a changeable parameter if response time is always too slow.

Issue L-3: Interest debt calculations use unadjusted asset amounts after token splits

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/23

Vulnerability Detail

When token splits occur, user asset amounts must be adjusted via `calculatePostSplitAmount()`. The `getInterestDebt()` function calculates interest based on a user's asset holdings but uses the raw storage value without split adjustments:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetPool.sol#L550-L560

```
function getInterestDebt(address user, uint256 cycle) public view returns (uint256
↪  interestDebt) {
    UserPosition storage position = userPositions[user];
@>  uint256 assetAmount = position.assetAmount;
    // ... calculate debt using unadjusted amount
    uint256 debt = Math.mulDiv(assetAmount, poolIndex - userIndex, PRECISION *
    ↪  reserveToAssetDecimalFactor);
    return debt;
}
```

This function is called by `calculateUserRequiredCollateral()` and `getUserCollateralHealth()` in the strategy contract. For example, after a 2:1 split, if a user initially had 200 tokens stored, he should only have 100 now, but `getInterestDebt()` still calculates interest for only 200 tokens instead of 100, overestimating the debt.

Impact

Users' interest debt is calculated incorrectly after token splits, causing collateral health checks to be inaccurate.

Recommendation

Apply split adjustments in `getInterestDebt()` before calculating debt.

Issue L-4: Protocol cannot handle reserve tokens with more decimals than asset tokens (18) [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/24

Vulnerability Detail

The protocol uses `reserveToAssetDecimalFactor` to convert between reserve token amounts (e.g., USDC) and asset token amounts (xTSLA). This factor is initialized based on decimal differences:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolStorage.sol#L96-L108

```
if (assetDecimals >= reserveDecimals) {
    reserveToAssetDecimalFactor = 10 ** uint256(assetDecimals - reserveDecimals);
} else {
@>  reserveToAssetDecimalFactor = 1;
    // Note: For the case where reserve has more decimals, additional handling
    // would be needed in conversion functions
}
```

When `reserveDecimals > assetDecimals`, the factor is incorrectly set to 1. The conversion functions rely on this factor:

```
function _convertAssetToReserve(uint256 assetAmount, uint256 price) internal view
↪   returns (uint256) {
    return Math.mulDiv(assetAmount, price, PRECISION * reserveToAssetDecimalFactor);
}

function _convertReserveToAsset(uint256 reserveAmount, uint256 price) internal view
↪   returns (uint256) {
    return Math.mulDiv(reserveAmount, PRECISION * reserveToAssetDecimalFactor,
    ↪   price);
}
```

With factor = 1 when it should be 10^(reserveDecimals - assetDecimals), all conversions produce incorrect results by orders of magnitude.

Impact

Pools with reserve tokens having more decimals than asset tokens will have completely broken accounting, causing massive calculation errors.

Recommendation

After discussion, since we don't want to support reserveTokens with more than 18 decimals, we can just revert during the `_initializeDecimalFactor()` function, just to be safe.

Issue L-5: uint256.max allowances overflow after token splits breaking deFi integrations [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/25

Vulnerability Detail

Many ERC20 integrations use `type(uint256).max` as an "infinite allowance" to avoid repeated approvals. The xToken contract converts visible amounts to raw storage amounts by dividing by `_splitMultiplier`:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/xToken.sol#L225-L229

```
function approve(address spender, uint256 amount) public override returns (bool) {
@>  uint256 rawAmount = Math.mulDiv(amount, PRECISION, _splitMultiplier);
    return super.approve(spender, rawAmount);
}
```

After a token split, `_splitMultiplier` increases. When any function attempts to convert `type(uint256).max`, it may face integer overflow.

The same issue affects `permit()` and `allowance()` view function, as both perform the same conversion.

Impact

DeFi protocols that granted infinite approvals (standard practice) can no longer interact with xToken after any split, breaking all integrations until approvals are manually reset.

Recommendation

Handle `type(uint256).max` as a special case that remains infinite:

```
function approve(address spender, uint256 amount) public override returns (bool) {
    if (amount == type(uint256).max) {
        return super.approve(spender, amount);
    }
    uint256 rawAmount = Math.mulDiv(amount, PRECISION, _splitMultiplier);
    return super.approve(spender, rawAmount);
}
```

The same for `allowance()`, `permit()` functions.

Also, in `transferFrom()`, if the allowance is uint256.max, don't deduct the allowance.

Issue L-6: Permit signatures grant value change after token splits [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/26

Vulnerability Detail

ERC20 permit signatures allow users to grant spending approvals via off-chain signatures. Users sign for a specific token amount representing a certain economic value. The xToken `permit()` function converts the signed amount using the current split multiplier:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/xToken.sol#L242

```
function permit(address owner, address spender, uint256 value, ...) public override
 ↪  {
@>  uint256 rawValue = Math.mulDiv(value, PRECISION, _splitMultiplier);
    super.permit(owner, spender, rawValue, deadline, v, r, s);
}
```

Example: A signs a signature B to approve 100 tokens. Then, token split happens (e.g. 5 -> 1) and token price boosts by 500%. Now, the same signature is worth 5x the value when it was signed.

Impact

Users who signed permits before splits unknowingly approve spending of significantly more economic value than intended.

Recommendation

Store the split multiplier at signing time and validate it hasn't changed, or invalidate all outstanding permits when splits occur by incrementing the nonce.

Discussion

bhargavaparoksham

Chatgpt flagged another issue which is valid, that in current permit implementation user signs value but we pass rawValue to super.permit. If they token had a split then this will fail because of the difference in what user signed & what we're passing.

I've pushed a fix for this as well.

Issue L-7: LP positions with zero collateral permanently block rebalancing [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/27

Vulnerability Detail

When an LP is liquidated, the liquidator receives a reward from the LP's collateral. If the collateral is insufficient to cover the full reward, all remaining collateral is transferred:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L462-L464

```
if (position.collateralAmount < rewardAmount) {
@>  transferAmount = position.collateralAmount;
}
```

This can leave an LP with non-zero `liquidityCommitment` but zero `collateralAmount` - a "ghost" position. No liquidator will liquidate this position further since there's no reward.

During settlement rebalancing via `rebalanceLP()`, if the LP owes a rebalance amount, the protocol calls `deductFromCollateral()`:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L415

```
function deductFromCollateral(address lp, uint256 amount) external
↪   onlyPoolCycleManager {
    LPPosition storage position = lpPositions[lp];
@>  if (position.collateralAmount < amount) revert InvalidAmount();
    // ... deduct logic
}
```

The transaction reverts when trying to deduct from zero collateral, blocking the entire cycle from progressing.

Impact

The protocol becomes permanently stuck when an LP reaches zero collateral. All rebalancing halts, preventing cycle transitions until additional collateral is added to the LP by treasury.

Recommendation

This seems to be by design, best to add explicit documentation that this case would be handled by protocol treasury so protocol doesn't halt.

Issue L-8: Code quality and minor issues [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/28

1. Redundant LP registration check in `resolveRequest()`

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L436

```
function resolveRequest(address lp) external onlyPoolCycleManager {
@>  if (!isLP[lp]) revert NotRegisteredLP();
    // ...
}
```

The function checks `isLP[lp]` but should use `isLPActive[lp]` for consistency and to prevent processing inactive LPs.

2. Missing validation for existing requests in LP liquidation

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L696

```
LPRequest storage request = lpRequests[lp];
if (request.requestType == RequestType.LIQUIDATE) {
    // Handle existing liquidation
}
// Missing: add check for request.requestType cannot be ADD_LIQUIDITY or
↪  REDUCE_LIQUIDITY.
```

While theoretically impossible (liquidation requires health == 1, while add/reduce require health != 1), adding an explicit check would improve safety.

3. Redundant `isLP()` check in LP rebalancing

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L293-L297 https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L396-L400 https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L464-L468

```
if (poolLiquidityManager.isLP(lp)) {
    lpLiquidityCommitment = poolLiquidityManager.getLPLiquidityCommitment(lp);
} else {
    lpLiquidityCommitment = 0;
}
```

The LP must still be registered, making this check unnecessary. The code can be simplified to directly `lpLiquidityCommitment = poolLiquidityManager.getLPLiquidityCommitment(lp);`.

4. Incorrect comment in `assetPool.deductInterest`

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#286

```
// Deduct interest from the pool and add to LP's collateral
if (lpCycleInterest > 0) {
    assetPool.deductInterest(lp, lpCycleInterest, false);
}
```

For `rebalancePool()` and `rebalanceLP()`, the comment is incorrect because, interest is added to `lpPositions[lp].interestAccrued`, and not directly added to LP's collateral.

For `forceRebalanceLP()`, the comment is incorrect because, interest goes to the protocol fee recipient, not to LP's collateral.

5. Unused `collateralAmount` field in user requests

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetPool.sol#L265

```
request.requestType = RequestType.DEPOSIT;
request.amount = amount;
@> request.collateralAmount = 0;  // Always set to 0, never read
request.requestCycle = currentCycle;
```

The `UserRequest.collateralAmount` field is always set to 0 and never used elsewhere in the codebase. Consider removing this unused field.

6. Division by zero risk when claiming assets with zero principal

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetPool.sol#L411

```
uint256 weightedOld = (oldPrincipal == 0) ? 0 : Math.mulDiv(oldPrincipal, uIndex,
↪  newPrincipal);
uint256 weightedNew = Math.mulDiv(assetAmount, interestIndex, newPrincipal);
```

If `newPrincipal == 0`, the second `Math.mulDiv` will cause division by zero. Add a special case: `newUserIndex = interestIndex` when `newPrincipal == 0`.

Discussion

bhargavaparoksham

fixed 1,3,4 & Acknowledging the rest.

Issue L-9: Yield is not attributed exactly and leads to skewed distribution [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/31

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Yield generated by the reserve does not account for debt accumulated by users which should not be counted as generating yield.

The updated yield code does not reflect users collateral which results in a skewed distribution.

Vulnerability Detail

Yield is generated by the reserve amount in the `AssetPool`, each users has a claim on a proportional amount. The interested paid to the `LiquidityManager` each cycle reduces the reserve amount.

When calculating yield we are not accounting for the debt the users has which would reduce their claim on reserve tokens.

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/a8e10d9294125053e2b9537baa63fb17f570c466/own-contracts/src/protocol/AssetPool.sol#L761-L765

with

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/a8e10d9294125053e2b9537baa63fb17f570c466/own-contracts/src/protocol/AssetPool.sol#L709-L718

The updated code with new yield functionality has a related issue

https://github.com/own-protocol/own-contracts/blob/7631cadf11271b3514313078125441e1631c738e/src/protocol/AssetPool.sol#L697-L712

The collateral is not precisely accounted for since the yield is distributed based on `assetAmount`. This leads to a skewed distribution as it does not reflect the true collateralization of users.

Impact

Users that has accumulate a lot of debt earn more yield than they should while new users (an users with low debt) earn less yield.

Skewed distribution of yield, users with lower collateralization receive less yield than expected. Users that deposit collateral without doing a deposit request will not earn any yield on their collateral.

Code Snippet

Tool Used

Manual Review

Recommendation

Account for debt when calculating users's true reserve and account for collateral deposits to exactly calculate users' claim on yield as both deposits and collateral earn yield.

Discussion

bhargavaparoksham

@hildingr

We agree with the core concern that the current yield distribution model does not allocate yield to collateral with full precision. However, the statement that “collateral does not earn yield” is not correct.

In the current implementation, collateral does accrue yield, but indirectly: yield is allocated based on the user’s share of the asset in the pool, rather than their exact collateral ratio. This means, in some cases, users with above-average collateralization might receive slightly less yield than the ideal proportional value, and users with lower collateralization receive slightly more. The effect is a distribution skew, not a total omission.

Why we accept this tradeoff now Bytecode limits. The AssetPool contract is ~150 bytes from the EVM limit. Precisely tracking collateral-based yield would require additional state and a structural redesign, which isn’t feasible at this stage.

Low-impact in practice. At launch, all pools use aUSDC, and collateral requirements are capped at 3%. With protocol yield set ~1% above Aave’s rate, the resulting yield misallocation is ~0.00006%, which we consider acceptable for V1.

Classification Given the scope of the protocol at launch, we want to mark this as acknowledged and request you to please mark this as low-impact.

Planned fix (V2) Once we have room to refactor beyond current bytecode constraints, we plan to implement a fully precise collateral-based yield model that attributes yield exactly according to each user’s current collateral after including the debt.

hildingr

> We agree with the core concern that the current yield distribution model does not allocate yield to collateral with full precision. However, the statement that “collateral does not earn yield” is not correct.

It is technically correct in an edge case: since a user can deposit collateral without doing a deposit request it is possible to have collateral but no assets deposit for many cycles. The user would not earn any yield.

Given that the issues are related but have different root cause and one is introduced in the updated code I figured it made sense to have two separate issues.

It is fine by me to have both in the same issue if you prefer that. Take a look at this gist https://gist.github.com/hildingr/dace0dcfa7c85a9fbbbc2ffd9c170a12

bhargavaparoksham

@hildingr

the gist looks good. We don't expect users to deposit collateral, without making the actual deposit amount. If they do they'll not accrue any yield. We can clearly write this in the docs etc, so that users don't do it by mistake.

As mentioned earlier, given the params with which we're going to launch, we're ok to make this trade off.

Note that we'll make sure, users are aware of this issue, by clearly explaining it in docs.

Considering the scope, please mark this issue as acknowledged & update the impact accordingly as low, under the specific condition that collateral requirements are below 3%. If needed you can also mention in the report that, in pools with higher collateral requirements this could have an impact on yield earned on collateral.

Issue L-10: Misleading comment when force rebalancing LP [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/32

Summary

The following comment indicates that the interest is the added to the LP's collateral while in reality it's taken as a penalty from it because this is a settlement case:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L456-459

```
// Deduct interest from the pool and add to LP's collateral
          if (lpCycleInterest > 0) {
              assetPool.deductInterest(lp, lpCycleInterest, true);
          }
```

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetPool.sol#L618-620

```
if (isSettle) {
          // During settlement, all interest goes to the protocol as penalty
          reserveToken.safeTransfer(poolStrategy.feeRecipient(), amount);
```

Impact

The overall readability is affected plus it can mislead somebody during internal/external reviews.

Tool Used

Manual Review

Recommendation

Change the comment accordingly.

Issue L-11: Missing initialization checks for several parameters [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/33

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

In the `initialize()` functions there are missing checks for some of the contracts.

Vulnerability Detail

There are several instances where not all of the initial parameters are validated:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/AssetPool.sol#L128-130

```
if (_reserveToken == address(0) || _assetOracle == address(0) ||
        _poolLiquidityManager == address(0) || _poolCycleManager == address(0))
        revert ZeroAddress();
```

As you can see here, there's missing `_assetPool` and `_poolStrategy` validation for non-zero addresses here. Same thing applies for the `_poolStrategy` and `_poolLiquidityManager` in the `PoolLiquidityManager` contract:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolLiquidityManager.sol#L121-124

```
if (_reserveToken == address(0) || _assetToken == address(0) || _assetPool ==
↪  address(0) ||
      _poolCycleManager == address(0) || _assetOracle == address(0)) {
      revert ZeroAddress();
  }
```

Impact

Potential initialization errors because of the missed validation.

Tool Used

Manual Review

Recommendation

Add the mentioned `address(0)` check for the all the contracts.

Discussion

bhargavaparoksham

As most of these contracts are already verified that they are not address(0) in the factory and since pool, cycleManager & liquidityManager are added via cloning. We acknowledge that adding it will be extra safe but we are opting not to include them, as the upstream guarantees already ensure correctness.

Issue L-12: Deviation between specification and the current implementation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/issues/34

Summary

Currently, there's a difference between the assumptions made by the official documentation and the code itself.

Vulnerability Detail

According to the documentation:

```
Each asset has a daily OHLC (Open, High, Low, Close) range published by the
↪  protocol’s oracle. LPs must submit a rebalance price within this range. But
↪  they get to choose where within the range to submit.
```

However, in the current code, it's only `Open` and `Close` prices that are allowed by the system:

https://github.com/sherlock-audit/2025-10-own-protocol-nov-3rd/blob/main/own-contracts/src/protocol/PoolCycleManager.sol#L576-580

```
function _validateRebalancingPrice(uint256 rebalancePrice) internal view {
    if (rebalancePrice < Math.min(cyclePriceOpen, cyclePriceClose) ||
        rebalancePrice > Math.max(cyclePriceOpen, cyclePriceClose)) {
        revert InvalidRebalancePrice();
    }
```

As you can see, the code allows LPs to only submit price that is within the maximum of the `Open` and `Close`. This potentially disallows LPs to submit unfair prices - but they can also submit the highest prices within `Open` and `Close` limit. And the most important thing is that it misaligns with what the official documentation states.

Impact

Invalid assumptions made by the specification. Users can't submit the prices in that exact range that disallows the following:

```
Submit a slightly higher or lower price on-chain (within bounds)
```

Tool Used

Manual Review

Recommendation

Correct either the code or the implementation for them to fully align.

Discussion

bhargavaparoksham

I've updated the docs, to clearly mention that LPs can only rebalance between open & close.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.