Hyperhyper Security Review

Pashov Audit Group

Conducted by: unforgiven, merlinboii, Udsen, jesjupyter, Bloqarl, Oualid, Matin

March 30th 2025 - April 12th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

This audit's fixes review was not fully completed at the time of this report and is expected to continue through subsequent follow-up security reviews by the team.

3. Introduction

A time-boxed security review of the **Hyperhyperfi/protocol** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Hyperhyper

Hyperhyper is a hyper-speculation application designed for speculators seeking high leverage with no liquidation risks within short time frames. The platform emphasizes ultra-short maturities, allowing for quick entries and exits, leading to more frequent trading opportunities and gains.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. 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

6. Security Assessment Summary

review commit hash - 212acfdcd4066334ff5ea06ce46b0bbca7ca212f

fixes review commit hash - a4430e759e445e3ca4b90444ae6720f1c5e90ffe

Scope

The following smart contracts were in scope of the audit:

- `Oracle`
- `AdminOperationalTreasury`
- `OperationalTreasury`
- `OperationalTreasuryStorage`
- `PoolDiamond`
- `LiquidityFacet`
- `PoolAdminFacet`
- `PoolViewFacet`
- `PositionInteractionFacet`
- `LiquidityRouter`
- `PoolErrors`
- `PoolHelpers`
- `PoolStorage`
- `PositionsManager`
- `Call`
- `Put`
- `AdminStrategy`
- `Strategy`
- `StrategyStorage`
- `StrategyView`
- `CoreTypes`
- `PositionParams`
- `MathUtils`
- `FullMath`
- `FixedPoint128`
- `Decimals`
- `BlackScholes`
- `SafeDecimalMath`
- `SignedSafeDecimalMath`
- `LPToken`
- `ERC721WithURIBuilderUpgradeable`
- `ETHUnwrapper`
- `IETHUnwrapper`
- `Fenwicks`
- `IV`

7. Executive Summary

Over the course of the security review, unforgiven, merlinboii, Udsen, jesjupyter, Bloqarl, Oualid, Matin engaged with Hyperhyper to review Hyperhyper. In this period of time a total of **56** issues were uncovered.

Protocol Summary

Protocol Name	Hyperhyper
Repository	https://github.com/Hyperhyperfi/protocol
Date	March 30th 2025 - April 12th 2025
Protocol Type	Options DEX

Findings Count

Severity	**Amount**
Critical	3
High	6
Medium	15
Low	32
Total Findings	**56**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Permanent lock due to missing unlock mechanism for options	Critical	Acknowledged
[C-02]	PLP transfer or balance updates do not update reward index	Critical	Acknowledged
[C-03]	Incorrect index reset in Fenwick tree after full withdrawal	Critical	Resolved
[H-01]	Option premium calculation underprices options	High	Resolved
[H-02]	Insufficient liquidity checks for PUT may cause exercise failures	High	Acknowledged
[H-03]	Inaccurate accounting in removeLiquidity() overstating liquidity	High	Resolved
[H-04]	Improper use of locked funds as liquidity in PositionInteractionFacet	High	Resolved
[H-05]	Vulnerability in PositionInteractionFacet slippage control due to spot price	High	Acknowledged
[H-06]	payOff() underflow in OperationalTreasury locks assets permanently	High	Resolved
[M-01]	Hardcoded slippage tolerance risks poor trading or DOS	Medium	Acknowledged
[M-02]	block.timestamp use in dex swap deadlines may cause poor trading	Medium	Resolved
[M-03]	Fee calculation uses full instead of net amount for pool impact	Medium	Acknowledged

[M-04]	Missing price feed validation in oracle contract	Medium	Acknowledged
[M-05]	Lp token transfer without lock data transfer	Medium	Resolved
[M-06]	Unrestricted LP burn enables price inflation to exploit deposits	Medium	Resolved
[M-07]	Rounding error in put option price reduces safety margin	Medium	Resolved
[M-08]	Weak signature validation allows multiple option mispricing vectors	Medium	Resolved
[M-09]	strategy contracts: exercise window ignored in premium calculation	Medium	Acknowledged
[M-10]	liquidityFacet: fenwick tree attack through multiple deposits	Medium	Resolved
[M-11]	liquidityFacet: unauthorized liquidity removal for other users	Medium	Resolved
[M-12]	operationalTreasury: assumes 1:1 USD value for base token	Medium	Resolved
[M-13]	connect() can be called externally in AdminStrategy causing DoS	Medium	Resolved
[M-14]	Reward Index Inflation Due to Rounding Up Can Trap Rewards	Medium	Resolved
[M-15]	Incorrect ERC20 transfer in withdrawFee prevents fee withdrawal	Medium	Resolved
[L-01]	Precision loss in fee calculation due to order of operations	Low	Acknowledged
[L-02]	Missing _disableInitializers() in implementation contract	Low	Acknowledged
[L-03]	Missing IERC165 support in	Low	Acknowledged

	supportsInterface()		
[L-04]	Changes in underlying token mid-operation may cause issues	Low	Acknowledged
[L-05]	Missing duplicate token validation in setTargetWeight	Low	Acknowledged
[L-06]	Role assignment mismatch in PoolAdminFacet initialization	Low	Acknowledged
[L-07]	Lingering permissions from uncleaned DEX swap token approvals	Low	Acknowledged
[L-08]	Dynamic maxDepositPerUser can stale Fenwick tree state	Low	Acknowledged
[L-09]	Risk corruption from uncontrolled id in setPositionsManager	Low	Acknowledged
[L-10]	Uninitialized maturityDate in Strategy.create()	Low	Acknowledged
[L-11]	Unbounded protocol fee issue in setProtocolFees()	Low	Acknowledged
[L-12]	Oracle: irreversible token configuration	Low	Acknowledged
[L-13]	PoolAdminFacet: removeToken() lacks protocol fee check	Low	Acknowledged
[L-14]	Strategy cannot be added after removal in AdminOperationalTreasury	Low	Acknowledged
[L-15]	payOff() blocked for option exercise when contract is paused	Low	Acknowledged
[L-16]	Inefficient payout transfer check in payOff()	Low	Resolved
[L-17]	Missing initializer guard in _init() function	Low	Acknowledged

[L-18]	Contract wallets incompatible with EIP-1271 signature verification	Low	Acknowledged
[L-19]	_computeMaturityDate uint32 timestamp limits protocol lifespan	Low	Acknowledged
[L-20]	Missing parent initializer call in __UUPSUpgradeable_init()	Low	Acknowledged
[L-21]	Missing slippage protection in strike price calculation for options	Low	Acknowledged
[L-22]	Omission of _refreshVirtualPoolValue() causes stale fees and PLP price	Low	Acknowledged
[L-23]	Token mismatch in operationalTreasury affects pool payouts	Low	Acknowledged
[L-24]	updateDexConfig() in poolAdminFacet ignores pair-specific DEX fees	Low	Acknowledged
[L-25]	Inconsistent decimal handling in liquidity calculation	Low	Acknowledged
[L-26]	End-of-day option buys vulnerable due to block.timestamp dependency	Low	Acknowledged
[L-27]	Token removal can be blocked by minimal deposits	Low	Acknowledged
[L-28]	Incorrect decimal conversion in LiquidityFaucet calculation	Low	Acknowledged
[L-29]	Incorrect EIP-712 data hashing implementation	Low	Acknowledged
[L-30]	Option buyers choose volatility to reduce premium	Low	Acknowledged

[L-31]	High gas use risk from Uniswap V3 quoter	Low	Acknowledged
[L-32]	Indexed keyword in events causes struct data loss	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Permanent lock due to missing unlock mechanism for options

Severity

Impact: High

Likelihood: High

Description

The protocol has a critical issue where liquidity can be permanently locked in the pool when options are not exercised within the specified window(`exerciseWindowDuration`) or have 0 `PNL`(at a loss). Here's the problematic flow:

```
// In PositionInteractionFacet.sol
function _lockLiquidity(Position memory pos) internal {
    PoolStorage.Layout storage strg = PoolStorage.layout();
    if (pos.opType == OptionType.CALL)
        strg.ledger.state.lockedToken[pos.uAsset] += pos.sizeUa;
    else if (pos.opType == OptionType.PUT)
        strg.ledger.state.lockedUSD += pos.sizeUSD;
}

// In StrategyView.sol
function isPayoffAvailable
  (uint256 positionID, address caller, address /*recipient*/ )
    external
    view
    virtual
    override
    returns (bool)
{
    return strg.setUp.treasury.manager().isApprovedOrOwner(caller, positionID)
        && _calculateStrategyPayOff(positionID) > 0
        && block.timestamp >= maturity
        && block.timestamp <= maturity + strg.setUp.base.exerciseWindowDuration;
}
```

The issue manifests in several ways:

- Liquidity is locked when an option is created.
- Liquidity is only unlocked through `exerciseOrLiquidatePosition`.
- `exerciseOrLiquidatePosition` is only called during `payoff`.
- `payoff` is only available when:
  - The option is profitable (`_calculateStrategyPayOff > 0`).
  - Within the exercise window (maturity to maturity + exerciseWindowDuration).

This creates scenarios where liquidity becomes permanently locked:

- Options not exercised within the time window.
- Options where the holder fails to exercise.

Additionally, this creates a potential attack vector:

1. The attacker opens multiple options with small premiums.
2. Intentionally doesn't exercise them.
3. Causes permanent locking of pool liquidity.

Recommendations

Implement an unlock mechanism for this situation.

[C-02] PLP transfer or balance updates do not update reward index

Severity

Impact: High

Likelihood: High

Description

Link

The reward calculation in `_getPendingRewards` depends on two key factors: the user's PLP token balance and their reward index (`usrRwdIndexQ128`).

```
function _getPendingRewards
      (address token, address user) internal view returns (uint256) {
        RewardInfo storage rewards = PoolStorage.layout
          ().ledger.state.rewardsByToken[token];
        ILPToken plp_ = ILPToken(PoolStorage.layout().setUp.base.plp);
        uint256 lpBalance = plp_.balanceOf(user);
        uint8 plpDecimals = IERC20Metadata(PoolStorage.layout
          ().setUp.base.plp).decimals();
        uint8 tokenDecimals = IERC20Metadata(token).decimals();

        uint256 rewardIndexQ128 = PoolStorage.layout
          ().ledger.state.usrRwdIndexQ128[token][user];
        rewardIndexQ128 = rewards.rewardsPerLPQ128 - rewardIndexQ128;

        return lpBalance.mulDiv(rewardIndexQ128, Q128).toDecimals
          (plpDecimals, tokenDecimals);
    }
```

While the PLP balance can change through various operations (`addLiquidity`, `withdrawLiquidity`, or direct token transfers), the reward index is only updated in specific scenarios. This could lead to incorrect accounting for the rewards during `PLP Token Transfers Or Other Balance Updates`(like `partial liquidity withdraw`).

```
// Current update points for usrRwdIndexQ128:
// 1. First deposit (_initRewardIndex)
// 2. Full exit (_initRewardIndex)
// 3. Manual claim (_claimRewards)
function _updateLpRewardIndex(address token, address user) internal {
    PoolStorage.layout().ledger.state.usrRwdIndexQ128[token][user] =
        PoolStorage.layout
          ().ledger.state.rewardsByToken[token].rewardsPerLPQ128;
}
```

In such cases, users can manipulate their rewards by transferring `PLP` tokens without updating their `usrRwdIndexQ128`:

- A malicious user could transfer a large amount of PLP tokens to themselves before claiming rewards.
- The calculation `lpBalance.mulDiv(rewardIndexQ128, Q128)` would use the inflated balance.

Recommendations

Implement a hook/accounting system to update the reward index whenever PLP token balances change.

[C-03] Incorrect index reset in `Fenwick tree` after full withdrawal

Severity

Impact: High

Likelihood: High

Description

When a user adds liquidity to the pool, the system triggers the `_updateLockData` function, which in turn calls the `Fenwicks._deposit` function to record the deposit amount and its associated lock duration (e.g., 7 days). The `Fenwicks._deposit` function updates a data structure known as the Fenwick tree to manage and lock the deposit amounts over time.

For a user's first deposit (`isDirty=true`), the system resets the tree values to avoid incorrect data affecting subsequent deposits:

```
function _fenwicksUpdate(
  FenwicksDatastorageself,
  uint256maxDepositEntries,
  uint256treeIndex,
  uint256value
)
    internal
{
    // when registering new values after full withdrawal, we need to overwrite
    // dirty values
    bool isDirty = self.sortedKeys.length == 1;
    while (treeIndex <= maxDepositEntries) {
        // when dirty overwrite, otherwise add
@1>
        isDirty ? self.fenwicksTree[treeIndex] = value : self.fenwicksTree[treeIndex]
@2>      uint256 nextIndex = treeIndex + (treeIndex & (~treeIndex + 1));
        if
        //(nextIndex > maxDepositEntries) break; // Stop if next update would exceed m
        treeIndex = nextIndex;
    }
}
```

The problem occurs because it only reset specific indices such as 1, 2, 4, 8, etc. and **does not reset other indices like 3, 5, 6, etc.** After a full withdrawal, if the user previously had deposits at indices 3 and 5, they remain untouched. As a result, the old values from these indices are mistakenly added to new deposit amounts. Users can unlock more amounts and in fact, get (`removeLiquidity`) the funds of other users in the pool.

Example Scenario:

1. Initial Deposit: User deposits at indices 1, 2, and 3 updating the Fenwick tree accordingly.
2. Full Withdrawal: The user withdraws all funds (after the unlock period).
3. First Deposit After Withdrawal: The user deposits again, with isDirty=true. Indices 1, 2, 4, 8, 16, etc., are updated with the new values (resetted).
4. Second Deposit: The user deposits again, updating index 2 with the new value.
5. Third Deposit: The user deposits again, and index 3 is updated. However, because the old value from the initial deposit (step 1) remains in index 3, a new deposit value will be added to the old value.

Result:

Because the old value from the first deposits is incorrectly retained in index 3, the user ends up unlocking more funds than they actually deposited. This could allow the user to withdraw assets that they did not originally contribute, and potentially withdraw funds belonging to other users.

Recommendations

Ensure that the system resets all indices in the Fenwick tree after a full withdrawal, not just specific indices like 1, 2, 4, 8, 16, etc.

8.2. High Findings

[H-01] Option premium calculation underprices options

Severity

Impact: High

Likelihood: Medium

Description

The option premium is calculated using the input `period`, while the actual duration until the maturity date is rounded up to the end of the day which is typically longer than the given `period`. This allows traders to get longer option durations than what they pay for.

The premium calculation in `Strategy._calculatePremium()` uses the input `period` in seconds to calculate the option price:

```
function _calculatePremium(
  uint256amount,
  uint256period,
  uint256preciseCurrentPrice,
  uint256strike,
  uint256iv
)
    internal
    view
    returns (uint256 premium)
{
    (uint256 callPremium, uint256 putPremium) =
@>      StrategyStorage.layout().setUp.base.bs.optionPrices
  (period, iv, preciseCurrentPrice, strike, 0);
```

However, the maturity date is rounded up to the end of the day in `PositionParams._computeMaturityDate()`:

```
function _computeMaturityDate(uint256 period) internal view returns (uint32) {
    uint256 maturityTimestamp = block.timestamp + period;
    maturityTimestamp -= (maturityTimestamp % 1 days);
    maturityTimestamp += 1 days - 1;
    return uint32(maturityTimestamp);
}
```

Therefore, the gap between the input `period` and the actual duration until maturity date can be up to `86399` seconds, which can be a significant difference for short-period positions.

Consider the following scenario:

1. Current time: `1743638400` (Thu Apr 03 2025 00:00:00).
2. User inputs period = 7 days = `604800` seconds.
3. Actual maturity: `1744329599` (Thu Apr 10 2025 23:59:59).

Input period: `604800` seconds Actual period: `1744329599 - 1743638400` = `691199` seconds (7 days + 86399 seconds ≈ 8 days).

The premium is calculated using the input period (`604800` seconds) in the Black-Scholes formula, resulting in a lower premium as it underestimates the time until maturity.

Below is an example output for a 7-day `CALL` option (3500 USD spot price, 10% relative strike, 48.9% IV). This results in approximately 23% underpricing for this `CALL` option position.

```
Logs:
  Period: 604800 sec
  Maturity: 1744329599
  Actual Period: 691199 sec
  --- Premium: period ---
  call: 8963759233891000000
  put: 358963759233891000000
  --- Premium: maturity - uint32(block.timestamp)---
  call: 11675104193168500000
  put: 361675104193168500000

  Premium difference: 2711344959277500000
```

Recommendation

Update the premium calculation to use the actual duration until the maturity date.

[H-02] Insufficient liquidity checks for `PUT` may cause exercise failures

Severity

Impact: High

Likelihood: Medium

Description

The protocol checks **total stablecoin value** when opening `PUT` positions and processing stablecoin withdrawals. However, **payouts during position exercise require a specific stablecoin (`pos.buyToken`)** to be available. This discrepancy can cause `PUT` positions to become unexercisable even when the overall pool appears solvent.

Consider the following scenarios: Assume:

- `OperationalTreasury.baseSetUp.token`: `USDC` (`buyToken`)
- Pool's stablecoin: `USDC` (listed & reward), `USDXL` (listed)
- Stable coin price is 1:1

Scenario 1: PUT position opens with 0 reserved for its payout token

Assume:

- `poolAmount[USDC]` = 0 USDC -> 0 USD
- `poolAmount[USDXL]` = 3500 USDXL -> 3500 USD
- `_getPoolValue(true, false)` = 0 + 3500 = 3500 USD

1. User opens `PUT` position, resulting in `lockedUSD` = 3500 USD
    - The check passes: `availableStable = 3500 ≥ lockedUSD:0 + pos.sizeUSD: 3500`
    - This allows to create the `PUT` position that will pay `USDC` if position is profit but there are no `USDC` in the pool.

```
function _checkEnoughLiquidity(
    //...
) internal pure {
    if (pos.opType == OptionType.CALL) {
        --- SNIPPED ---
    } else if (pos.opType == OptionType.PUT) {
@>      if (lockedUSD + pos.sizeUSD > availableStable) {
            revert NotEnoughLiquidityPut
              (pos.sizeUSD, availableStable, lockedUSD);
        }
    }
}
```

2. User attempts to exercise their `PUT` position, but if at that time the `poolAmount[USDC]` still holds 0 `USDC`, the `_payout()` reverts.

```
function _payout(
  Positionmemorypos,
  PositionClosememoryclose
) internal returns (uint256 pnl
    --- SNIPPED ---
@>  strg.ledger.state.poolAmount[pos.buyToken] -= pnl;
    _doTransferOut(pos.buyToken, msg.sender, pnl);
}
```

Scenario 2: LP withdrawal drains payout token after the position opened

Assume:

- `poolAmount[USDC]` = 2000 USDC -> 2000 USD
- `poolAmount[USDXL]` = 3500 USDXL -> 3500 USD
- `_getPoolValue(true, false)` = 2000 + 3500 = 5500 USD

1. User opens `PUT` position, resulting in `lockedUSD` = 3500 USD
    - The check passes: `availableStable = 5500 ≥ lockedUSD:0 + pos.sizeUSD: 3500`

2. Liquidity provider removes their liquidity and requests token output as `USDC`
    - LP amount worth 2000 `USDC` equivalent
    - `_checkWithdrawlImpact()` checks: `newUSDValuation = 5500 - 2000 = 3500 USD`, which is `newUSDValuation: 3500 >= lockedUSD: 3500`
    - The process allows to withdraw 2000 `USDC` amount
    - `poolAmount[USDC]` = 2000 - 2000 = 0 `USDC`

```
function _checkWithdrawlImpact(
    //...
) internal view {
    --- SNIPPED ---
    else if (strg.setUp.assets.stablecoins.contains(_token)) {
        uint8 decimals = IERC20Metadata(_token).decimals();
        uint256 withdrawUSD = (_amount * _assetPrice).toDecimals
          (decimals + 8, 18);
@>      uint256 newUSDValuation = _getPoolValue(true, false) - withdrawUSD;

@>      if
  (newUSDValuation < strg.ledger.state.lockedUSD) revert PoolErrors.InsufficientPoolAm
    }
}
```

3. User attempts to exercise their `PUT` position, but `poolAmount[USDC]` = 0 `USDC`, so the `_payout()` reverts.

Recommendation

Track per-buyToken (stablecoin) locked amounts and validate against individual balances in both `PUT` position opening and withdrawal impact checks.

[H-03] Inaccurate accounting in `removeLiquidity()` overstating liquidity

Severity

Impact: Medium

Likelihood: High

Description

`LiquidityFacet.removeLiquidity()` incorrectly updates the pool's token balance by only deducting the net amount (`netOutAmount`) while separately tracking the fee amount (`pFee`).

This creates a discrepancy because the full withdrawal amount (`outAmount = netOutAmount + pFee`) is taken from the pool's token balance, but only `netOutAmount` is deducted from the accounting balance in `poolAmount`.

Consider the following example:

1. User calls `LiquidityFacet.removeLiquidity()` with `_lpAmount` LP tokens.
2. Protocol calculates:
    - `outAmount` = total tokens to withdraw
    - `pFee` = protocol fee
    - `netOutAmount = outAmount - pFee` (total tokens to withdraw after fee)
3. Protocol updates state:

```
function _removeLiquidityFromPool(
    ...
) internal {
    --- SNIPPED ---
@>  strg.ledger.protocolFee[_tokenOut] += pFee;
@>  strg.ledger.state.poolAmount[_tokenOut] -= netOutAmount;
    _refreshVirtualPoolValue();
}
```

4. The actual token extracted from the pool is `netOutAmount + pFee: outAmount` but only `netOutAmount` is deducted from `poolAmount`. This leads to `poolAmount` being inflated by `pFee`.

Recommendation

As the `removeLiquidity` function is to deduct the full amount from `poolAmount` and keep some fees in `protocolFee`, the `poolAmount` should be deducted by `outAmount: netOutAmount + pFee`.

[H-04] Improper use of locked funds as liquidity in `PositionInteractionFacet`

Severity

Impact: Medium

Likelihood: High

Description

When someone exercises a Call option, they receive their profit (`pnl`) in the `buyToken`. The contract should only use available (not locked) tokens for this payout, or swap assets if needed. The `_payout()` function attempts to use unclaimed LP rewards (premiums) to cover option payouts:

```
function _payout(
      Positionmemorypos,
      PositionClosememoryclose
    ) internal returns (uint256 pnl
        --snip--

                uint256 availablePayoutToken = strg.ledger.state.rewardsByToken[pos.b

        pnl = close.pnl;

        if (pos.opType == OptionType.CALL && availablePayoutToken < pnl) {
            uint256 swapOut = pnl - availablePayoutToken;
            _swapAssetToAsset(pos.uAsset, pos.buyToken, swapOut);
        }
        --snip--

        strg.ledger.state.poolAmount[pos.buyToken] -= pnl;
        _doTransferOut(pos.buyToken, msg.sender, pnl);
    }
```

However, this approach is fundamentally flawed because:

1. **Rewards Are Already Distributed**: The reward system works by updating indexes, meaning rewards are considered distributed to users as soon as they're accrued, even if not yet claimed.
2. **No Actual Token Reservation**: The contract mistakenly treats `rewardsByToken[pos.buyToken].totalRewards` as available liquidity when these funds are already allocated to users and some users may even have claimed their rewards.

As a result, when paying out Call options, the contract incorrectly uses part of the `buyToken` balance (the amount equal to totalRewards) to cover the user's profit (pnl). However, it does not account for the portion of `buyToken` that is locked and reserved for fulfilling Put option payouts.

Recommendations

Multiple solutions can be considered to ensure proper handling of payouts of Call options:

1. Avoid using the available `buyToken`s for payouts of Call options; instead, swap the `uAssets` as needed.
2. Track both available and locked balances for `buyToken`.
3. Incorporate checks for `total stablecoin value` and `lockedUsd`, and consider swapping stablecoins to the `buyToken` as part of the payout process.

[H-05] Vulnerability in `PositionInteractionFacet` slippage control due to spot price

Severity

Impact: High

Likelihood: Medium

Description

The `_swapAssetToAsset` function uses the spot price from the Uniswap pool to calculate the `amountInMaximum` when performing token swaps:

```
function _swapAssetToAsset(
        address assetIn,
        address assetOut,
        uint256 amountOut // in assetOut decimals
    ) internal returns (uint256 amountIn) {
        PoolStorage.Layout storage strg = PoolStorage.layout();
        PoolStorage.Dex memory dex = strg.setUp.base.dex;
        uint160 sqrtPriceLimitX96 = 0;


            tokenIn: assetIn,
            tokenOut: assetOut,
            amount: amountOut,
            fee: dex.dexFee,
            sqrtPriceLimitX96: sqrtPriceLimitX96
        });
        (uint256 amountInMaximum,,,) = IQuoterV2
          (dex.dexQuoter).quoteExactOutputSingle(quoteParams);
        amountInMaximum = _applySlippage(amountInMaximum);

        IERC20Metadata(assetIn).safeIncreaseAllowance
          (dex.dexRouter, amountInMaximum);

        --snip--
    }
```

This exposes the contract to front-running and sandwich attacks, as malicious actors can manipulate the spot price by executing their own transactions. This occurs as follows:

1. An attacker can observe the transaction flow of `exerciseOrLiquidatePosition` and subsequently the call to `_swapAssetToAsset`.

2. The attacker can frontrun the transaction and manipulate the spot price in the Uniswap pool.
3. The contract calls `quoteExactOutputSingle` to determine the amount of the input token (`amountInMaximum`) required to meet the output token amount (`amountOut`). This calculation is based on the manipulated spot price.
4. The contract applies slippage tolerance (`_applySlippage`), which allows for a margin of error in the swap calculation. However, since the spot price is already manipulated to favor the attacker, applying slippage further benefits the attacker.

As a result, the attacker can manipulate the spot price to their advantage, ensuring that the contract swaps tokens at a favorable rate for the attacker, resulting in a loss for the contract.

Recommendations

Consider using a more secure method of price calculation, such as using an oracle or TWAP source to prevent price manipulation from affecting the contract.

[H-06] `payOff()` underflow in `OperationalTreasury` locks assets permanently

Severity

Impact: High

Likelihood: Medium

Description

The `payOff()` function in the `OperationalTreasury` contract calculates `feesUSD` based on the full position size and current price, without considering the actual profit (`pnl`). Specifically:

- `feesUSD = positionAmount × FeePercentage × CurrentPrice`
- `pnl = positionAmount × (CurrentPrice - StrikePrice)`

This means that for positions where `CurrentPrice` is only slightly above `StrikePrice`, the calculated `feesUSD` may exceed the `pnl`. In such cases,the `netAmount = pnl - feesUSD` line will **underflow**:

```
function payOff(
      uint256positionID,
      addressaccount
    ) external override nonReentrant whenNotPaused {
        --snip--
@2>     IPositionInteraction(baseSetUp.pool).exerciseOrLiquidatePosition(close);

        uint256 feesUSD = _calculateFeesUSD(uAsset, uint256(amount)).toDecimals
          (18, baseSetUp.token.decimals());
@1>     uint256 netAmount = pnl - feesUSD;

        // fees deduced from PnL
        strg.ledger.collectedFees[address(baseSetUp.token)] += feesUSD;
        --snip--
    }
```

When `payOff()` reverts due to the underflow at `@1>`, the call to `exerciseOrLiquidatePosition()` at `@2>` is not executed. As a result, the **underlying assets in the pool remain locked forever**.

Recommendations

Cap the fees to be no greater than the actual pnl to prevent underflow and allow position closure:

```
uint256 feesUSD = _calculateFeesUSD(...);
if (feesUSD > pnl) {
    feesUSD = pnl;
}
uint256 netAmount = pnl - feesUSD;
```

8.3. Medium Findings

[M-01] Hardcoded slippage tolerance risks poor trading or DOS

Severity

Impact: Medium

Likelihood: Medium

Description

In the `PositionInteractionFacet.sol` contract, the slippage tolerance is hardcoded to 2% in the `_applySlippage` function:

```
function _applySlippage(uint256 amountIn) internal pure returns (uint256) {
    // 2% slippage from quoter estimation
    return amountIn + (amountIn * 20_000) / DEX_PRECISION;
}
```

This implementation presents several critical issues:

1. The fixed 2% slippage tolerance doesn't account for varying market conditions and liquidity pool depths.
2. During periods of high market volatility, this fixed tolerance may be insufficient, leading to failed transactions (DOS).
3. For highly liquid pools, 2% slippage is unnecessarily high, resulting in worse execution prices for users.
4. For low liquidity pools, 2% might be too low, causing transactions to fail when they could have succeeded with a higher tolerance.

Recommendations

Allow the tolerance to be set by the admin/owner.

[M-02] `block.timestamp` use in dex swap deadlines may cause poor trading

Severity

Impact: Medium

Likelihood: Medium

Description

In the PositionInteractionFacet.sol contract, when executing DEX swaps, the `deadline` parameter is set to `block.timestamp + 30 minutes`:

```
tokenIn: assetIn,
    tokenOut: assetOut,
    fee: dex.dexFee,
    recipient: address(this),
    deadline: block.timestamp + 30 minutes,  // Unnecessary buffer
    amountOut: amountOut,
    amountInMaximum: amountInMaximum,
    sqrtPriceLimitX96: sqrtPriceLimitX96
});
```

This implementation is problematic for several reasons:

1. The `30-minute` buffer is unnecessary since the transaction will be executed at the current `block.timestamp` anyway.
2. This extended deadline window allows malicious MEV bots to potentially withhold the transaction and execute it at a less optimal time.
3. This implementation doesn't provide any real protection against transaction delays since the actual execution time is determined by the network's block production.

Recommendations

Replace the current deadline implementation with the user input `deadline`.

[M-03] Fee calculation uses full instead of net amount for pool impact

Severity

Impact: Medium

Likelihood: Medium

Description

Link

In the `_calcAddLiquidity` function, there exists a discrepancy between the amount used for fee calculation(`_amountIn`) and the actual amount added to the pool(`netAmount`). The issue occurs in the following sequence:

```
// In _calcAddLiquidity:
valueChange = (_amountIn * tokenPrice).toDecimals
  (tokenDecimals + provDecimals, 18);
(, pFee) = _calcFeeRate
  (_token, _amountIn, valueChange, baseSetUp.addRemoveLiquidityFee, true);
netAmount = _amountIn - pFee;
```

The `valueChange` is calculated using the full `_amountIn` before any fees are deducted. This value is then used in `_calcFeeRate` to determine the pool's token ratio impact through the `_calcDelta` function. However, the actual amount being added to the pool is `netAmount` (which is `_amountIn - pFee`).

This creates a mismatch between:

1. The impact assessment used for fee calculation (based on the full amount).
2. The actual impact on the pool (based on net amount after fees).

For example, if a user adds `1000` Token with a 3% fee:

- The fee calculation assesses the pool impact using `1000` Token.
- But only `970` token is actually added to the pool.
- This leads to incorrect fee calculations as the actual pool impact is less than what was used to calculate the fee.

Recommendations

Charge Liquidity Provider with `_amountIn+pFee`.

[M-04] Missing price feed validation in oracle contract

Severity

Impact: Medium

Likelihood: Medium

Description

The `Oracle` contract's `_getLastPrice` function retrieves price data from oracle price feeds without performing critical validation checks, which could lead to stale or invalid price data being used in the protocol.

Link: https://github.com/Hyperhyperfi/protocol/blob/212acfdcd4066334ff5ea06ce46b0bbca7ca212f/src/core/oracle/Oracle.sol#L96-L101

```
function _getLastPrice(address token) internal view returns
      (uint256 price, uint256 timestamp) {
        (
          ,
          int256answer,
          /*uint256startedAt*/,
          uint256updatedAt,

        ) = tokenOracle[token].latestRoundData(

        price = answer.toUint256();
        timestamp = updatedAt;
    }
```

This vulnerability exists in the following areas:

- Staleness Check:
  - The function does not verify if the price data is fresh by comparing the `updatedAt` timestamp with the current `block.timestamp`.
  - Stale price data could be used for critical protocol operations.

- Price Validity:
  - No validation of the answer value to ensure it's not going to go below `minAnswer` defined by the `breaker`.

- L2-Specific Concerns:
  - When deployed on L2 networks, the contract lacks sequencer uptime validation (To be validated).

Recommendations

Implement comprehensive price feed validation in the `_getLastPrice` function.

[M-05] Lp token transfer without lock data transfer

Severity

Impact: Medium

Likelihood: Medium

Description

The `LPToken` contract allows LP tokens to be freely transferred between addresses. However, the protocol tracks liquidity lock data in the pool contract using the original depositor's address, not the token itself. When LP tokens are transferred to a new address, the lock data remains with the original depositor, making it impossible for the new holder to redeem the underlying assets.

Proof of Concept:

1. Alice deposits 1000 USDC and receives 1000 LP tokens with a 7-day lock:
   - Alice's sum lock: 1000 LP.

2. Alice transfers 500 LP tokens to Bob:
    - Bob's sum lock: 0 LPs.
    - Alice's sum lock still: 1000 LP.

3. After 7 days, Bob tries to redeem his 500 LP tokens:
    - The transaction reverts because `500 LP > _getUserUnlockedLP(bob): 0` as Bob has no lock data.

4. The LP tokens become locked as they can't be redeemed by either Alice (no longer owns tokens) or Bob (no lock data) unless Bob transfers the remaining 500 LP tokens back to Alice.

Recommendation

Consider disallowing LP token transfers entirely, or redesigning the system to manage lock data in a transferable way.

[M-06] Unrestricted LP burn enables price inflation to exploit deposits

Severity

Impact: High

Likelihood: Low

Description

The `LPToken` contract allows unrestricted burning of LP tokens via `burn()` and `burnFrom()`. This enables an attack where the first depositor manipulates the LP price by burning nearly all of their LP tokens. This skews the pool state such that any subsequent deposit may mint **less or 0 LP tokens**, either causing a denial of service (if slippage checks are used) or silent fund loss (if not).

```
function _calcAddLiquidity() internal view returns
  (uint256 netAmount, uint256 pFee, uint256 lpAmount) {
    --- SNIPPED ---
    if (lpSupply == 0) {
        lpAmount = netAmount * tokenPrice;
        lpAmount = lpAmount.toDecimals(tokenDecimals + 8, plpDecimals);
    } else {
@1>     lpAmount = netAmount.mulDiv(tokenPrice * lpSupply, _getPoolValue
  (false, false) /*18, USD*/);
@2>     lpAmount = lpAmount.toDecimals(tokenDecimals + 8, plpDecimals);
    }
}
```

```
//> src/utils/tokens/lp/LPToken.sol
function burn(uint256 value) public override(ILPToken, ERC20Burnable) {
    ERC20Burnable.burn(value);
}

function burnFrom(address _account, uint256 _amount) public override
  (ILPToken, ERC20Burnable) {
    ERC20Burnable.burnFrom(_account, _amount);
}
```

Consider the following scenario: 0. Initial states: - ETH Oracle price: 1500e8 (8 decimals, 1500 USD). - USDC Oracle price: 1e8 (8 decimals, 1 USD). - LP token decimals: 18. - 0% fee.

1. Bob attempts to add 1 ETH liquidity, which is equivalent to 1500 USD.
2. Alice front-runs Bob's deposit with 1501 USDC, which is equivalent to 1501 USD:
    - LP mint amount: `1501e6 * 1e8 = 1501e14` -- to 18 decimals --> `1501e14 * 1e(18-14) = 1501e18 LPs`.
    - PoolValue: 1501e18 ≈ 1501 USD.
3. Alice burns all but 1 wei of LP tokens, so now 1 LP is worth 1501 USD.
4. Bob's transaction gets executed:
    - PoolValue: 1501e18 ≈ 1501 USD.
    - LP mint amount: `(1e18 * 1500e8) / 1501e18 = 99999999` -- to 18 decimals --> `99999999 / 1e(26-18) = 99999999 / 1e8 = 0 (rounding down)`.
    - Due to rounding down in `lpAmount` calculation, it causes the `lpAmount` to be 0.
5. Bob also specifies the slippage `_minLpAmount`. This attack will create a DoS if `_minLpAmount` is specified. If not, this attack will steal Bob's deposits.

Recommendation

Restrict burning to protocol roles: Pool contract for removing liquidity or disallowing public `burn()` and `burnFrom()` if `lpSupply` is less than the minimum LP supply.

[M-07] Rounding error in put option price reduces safety margin

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol applies the same strike price rounding logic to both `CALL` and `PUT` options. While this works in favor of the protocol for `CALL` options, it reduces safety for `PUT` options.

Specifically, the strike price is first computed by applying a relative discount to the current price (for PUTs), and then passed to `_roundPrice()`, which rounds **upward** if the remainder exceeds the midpoint. This pushes the strike **closer to or into the money**, reducing the out-of-the-money (OTM) buffer intended for PUT options.

```
function _calculateStrike(
  uint256preciseCurrentPrice,
  uint256relativeStrike
) internal view returns (uint256
    uint256 calculatedStrike = preciseCurrentPrice - (
      (preciseCurrentPrice * relativeStrike) / 100);
@>  return _roundPrice(calculatedStrike);
}
```

```
function _roundPrice(uint256 price) internal view returns (uint256) {
    RoundingOption currentRoundingOption_ = StrategyStorage.layout
      ().setUp.base.currentRoundingOption;
    uint256 remainder;
    uint256 roundedDown;
    if (currentRoundingOption_ == RoundingOption.NEAREST_HUNDRED) {
        remainder = price % (100 * 1e18);
        roundedDown = price - remainder;
@>      if (remainder >= 50 * 1e18) return roundedDown + (100 * 1e18);
        else return roundedDown;
    } else if (currentRoundingOption_ == RoundingOption.NEAREST_TEN) {
        remainder = price % (10 * 1e18);
        roundedDown = price - remainder;
@>      if (remainder >= 5 * 1e18) return roundedDown + (10 * 1e18);
        else return roundedDown;
    } else {
        return price;
    }
}
```

Consider this scenario with `NEAREST_HUNDRED` rounding: **Scenario 1:**

- Spot price: 1000 USD.
- Relative strike: 5%.
- Raw PUT strike = 950 USD.
- Rounded strike = 1000 USD (ATM instead of OTM).

Scenario 2:

- Spot price: 3500 USD.
- Relative strike: 10%.
- Raw PUT strike = 3150 USD.
- Rounded strike = 3200 USD (300 (~8.25% relative strike) OTM instead of 350 OTM).

In both cases, upward rounding reduces the distance between the strike and spot price for PUT positions.

Recommendation

Use different rounding directions for `CALL` and `PUT` options to reflect their opposite risk implications:

- For `CALL` options, keep rounding strike prices up, since that makes them more out-of-the-money and benefits the protocol.
- For `PUT` options, round down instead, so the strike stays further from the current price and doesn't unintentionally favor the trader.

[M-08] Weak signature validation allows multiple option mispricing vectors

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol's signature validation mechanism enables option mispricing through period manipulation and stale signatures:

1. **Period and Maturity Mismatch**

```
function _hash(...) internal pure returns (bytes32) {
@>  return keccak256(abi.encode
  (iv, uAsset, oType, maturityDate, relativeStrike));
}

function iv(
  uint48period,
  OptionTypeoType,
  uint256strikePrice
) external view returns (uint256 value
@>  value = _ivValues[period][oType][strikePrice]; // IV is queried based on
// `period`, `oType` and `strikePrice`
}
```

2. **Missing Signature Deadline**

```
function _recoverSigner(...) internal view returns (address) {
@>  address recoveredSigner = _recoverSigner
  (sig, iv, strategy, period._computeMaturityDate(), relativeStrike);
}
```

This enables two primary attack vectors:

1. **Period Manipulation**

- User requests signature for a 14-day period on April 3 (maturity: April 17 23:59).
- On April 10, the user executes the option with a 7-day period (maturity: April 17 23:59, results in the same maturity).
- The process validates this position with IV intended for a 14-day period, mispricing the option.

2. **Stale Strike Price**

- User requests signature for a 10% relative strike at a spot price 3500 USD(strike = 3850 USD).
- User waits until the market moves to some spot price, ie 4200 USD (new strike = 4620 USD).
- User executes the option with a stale signature, still using the IV for strike 3850 USD.
- The protocol applies an outdated IV, mispricing the option.

Recommendation

Include all critical parameters and enforce signature expiration validation:

```
function _hash(...) internal pure returns (bytes32) {
    return keccak256(abi.encode(
        iv,
        uAsset,
        oType,
+       period,
        maturityDate,
        relativeStrike,
+       deadline
    ));
}
```

[M-09] `strategy` contracts: exercise window ignored in premium calculation

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol currently allows options to be exercised within a **window after the maturity date** (i.e., `exerciseWindowDuration`):

```
function isPayoffAvailable
      (uint256 positionID, address caller, address /*recipient*/ )
        external
        view
        virtual
        override
        returns (bool)
    {
        --snip--
        return strg.setUp.treasury.manager().isApprovedOrOwner
          (caller, positionID)
            && _calculateStrategyPayOff
              (positionID) > 0 && block.timestamp >= maturity
@>
            && block.timestamp <= maturity + strg.setUp.base.exerciseWindowDuration;
    }
```

However, the **premium calculation** does not account for this extended exercise window. This creates an inconsistency, as the price at the maturity date should be used to calculate the value of the option, but the current price after the maturity date is being used instead. An option buyer can wait for the option price to increase after the maturity date, exercise his option at a more favorable price, and gain a positive profit and loss (PnL).

Note: The protocol uses **European Options** and their pricing model, which assumes options should only be exercised at the **maturity date price**. However, in the current strategy contracts, exercise is allowed within a window of time (`exerciseWindowDuration`) after the maturity date. Options are paid off based on the current price, not the maturity date price.

Note: Reducing the exercise window duration will increase the risk of users missing the opportunity to call `payOff` and losing their potential profit.

Example Scenario (based on tests and scripts numbers):

- **Option period** = 7 days
- **Exercise window duration** = 3 days (extension of 50% of the original period)

A user pays the premium for a 7-day option, but after the maturity date, they are allowed an additional 3-day window to exercise the option. During this period, the user can wait for the price to increase and then exercise the option, maximizing their profit. However, the premium is not adjusted to account for this extra time.

Recommendation

To resolve this issue, consider the following alternatives:

1. **Always exercise at the maturity date price**: Use historical price oracles to ensure that options are exercised at the maturity date price.

2. **Include the exercise window duration in the premium calculation**: Adjust the premium to fairly reflect the time extension provided by the `exerciseWindowDuration`, ensuring the premium remains consistent with the option's original value and removing the potential for manipulation.

The alternative one is more aligned with *the European options pricing model* and also eliminates the need for having a strict window to call `payOff` to exercise the options.

[M-10] `liquidityFacet`: fenwick tree attack through multiple deposits

Severity

Impact: Medium

Likelihood: Medium

Description

In the current implementation of the `addLiquidity()` function, a malicious user can exploit the Fenwick Tree structure by adding liquidity multiple times on behalf of another user, effectively filling the tree and potentially preventing legitimate deposits from occurring. The Fenwick Tree has a `maxDepositEntries` limit, and once this limit is reached, no more entries can be added. This enables griefing behavior, where an attacker could manipulate the system by adding multiple deposits for a single user, thus locking out legitimate users from depositing liquidity until the tree is reset.

```
function _deposit(
      FenwicksDatastorageself,
      uint256maxDepositEntries,
      uint256lockDuration,
      uint256amount
    )
        internal
    {
        uint256 treeIndex = _registerKey
          (self, maxDepositEntries, block.timestamp + lockDuration);
        _fenwicksUpdate(self, maxDepositEntries, treeIndex, amount);
        self.totalMintedPLP += amount;
    }

    function _registerKey(
      FenwicksDatastorageself,
      uint256maxDepositEntries,
      uint256newTimestamp
    )
        internal
        returns (uint256 treeIndex)
    {
        uint256 keyCount = self.sortedKeys.length;

@>      if (keyCount >= maxDepositEntries) revert Fenwicks_TooManyKeys();

        self.sortedKeys.push(newTimestamp);
        self.keyToIndex[newTimestamp] = ++keyCount;

        treeIndex = self.keyToIndex[newTimestamp];
    }
```

Although users can withdraw all liquidity to reset the tree, this action can only be performed after the lock duration has passed, which may lead to prolonged periods where malicious behavior can persist.

Recommendations

Check `msg.sender` in add liquidity to prevent unauthorized access.

[M-11] `liquidityFacet`: unauthorized liquidity removal for other users

Severity

Impact: Low

Likelihood: High

Description

The `removeLiquidity()` function currently allows any user to call the function and remove liquidity on behalf of another user, as it does not properly restrict access to the user performing the action. This could lead to unauthorized removal of liquidity from another user's account, potentially resulting in loss of assets, especially in scenarios where malicious actors attempt to exploit the function.

```
function removeLiquidity
      (address _tokenOut, uint256 _lpAmount, uint256 _minOut, address _to)
        external
        override
        nonReentrant
        whenNotPaused
    {
        _requireAmount(_lpAmount);
        _validateAsset(_tokenOut);

        if (_lpAmount > _getUserUnlockedLP
          (_to)) revert PoolErrors.LiquidityLocked();

        // remove liquidity
        (
          ,
          uint256netOutAmount,
          uint256pFee,
          uint256tokenOutPrice
        ) = _calcRemoveLiquidity(_tokenOut, _lpAmount
        _removeLiquidityFromPool
          (_minOut, netOutAmount, _tokenOut, tokenOutPrice, pFee);

        uint256 rewardsTokenOut = _isFullyExiting
          (_to, _lpAmount) ? _claimAllRewards(_to, _tokenOut) : 0;

        // PLP management
        _updateLockData(false, _lpAmount, _to);
        _exitAndBurnPLP(_lpAmount, _to);

        _doTransferOut(_tokenOut, _to, netOutAmount + rewardsTokenOut);
        _emitPLPPrice();
        emit LiquidityRemoved
          (_to, _tokenOut, _lpAmount, netOutAmount, rewardsTokenOut, pFee);
    }
```

Recommendations

Implement access control to ensure that only the user who owns the liquidity can remove their liquidity. This can be done by checking if the `_to` address is the same as the caller (`msg.sender`) before proceeding with the liquidity removal.

[M-12] `operationalTreasury`: assumes 1:1 USD value for base token

Severity

Impact: High

Likelihood: Low

Description

In `OperationalTreasury`, the protocol calculates fees and premium amounts in USD, but it assumes that the `baseSetUp.token` (buy token) value is always equal to 1 USD. This assumption can lead to a wrong collection of funds if the token is valued at less than 1 USD, or worse if it becomes depegged.

`_takeFeesLockLiquidity` function:

```
function _takeFeesLockLiquidity
  (Position memory pos, uint256 amount, uint256 premium, uint8 tokenDecimals)
    internal
    returns (uint256 fees)
{


    // fees and transfer funds here
    {
        fees = _calculateFeesUSD(pos.uAsset, amount).toDecimals
          (18, tokenDecimals);

        // fees added on top of premium
        OperationalTreasuryStorage.layout().ledger.collectedFees[address
          (baseSetUp.token)] += fees;

        // funds transfer from buyer to treasury. Premium transferred to pool,
        // fees stay in treasury
        baseSetUp.token.safeTransferFrom(msg.sender, address
          (this), premium + fees);
    }

    // lock liquidity in pool
    {
        baseSetUp.token.safeIncreaseAllowance(baseSetUp.pool, premium);
        IPositionInteraction(baseSetUp.pool).openPosition(pos);
    }
}
```

In the current implementation:

- Fees are calculated in USD with 18 decimals.
- Then converted to token decimals (e.g., 6 or 18) using .toDecimals().
- The amount of base token is transferred from msg.sender.
- But the actual exchange rate between the base token and USD is not accounted for. This leads to incorrect fees and premium amounts being transferred from the user, especially if the token loses its peg or fluctuates significantly.

Note: Same issue happens in `payoff`, The `pnl` amount is calculated in USD, but finally the amount of the base token is transferred to the user.

Recommendations

Consider using base token (buy token) price and value in calculations.

[M-13] `connect()` can be called externally in `AdminStrategy` causing DoS

Severity

Impact: Medium

Likelihood: Medium

Description

The `connect()` function in the `AdminStrategy` contract is externally callable and lacks access control, which allows any address to call it before the official initialization via `AdminOperationalTreasury`. Once called, it sets the `setUp.treasury` field, and any subsequent legitimate call (including the one from `init()`) will revert due to the `TreasuryAlreadySet()` check.

```
function connect() external override {
        StrategyStorage.SetUp storage setUp = StrategyStorage.layout().setUp;
        IOperationalTreasury treasury_ = IOperationalTreasury(msg.sender);
        if (address(setUp.treasury) != address(0)) revert TreasuryAlreadySet();
        setUp.treasury = treasury_;
    }
```

This creates a denial-of-service (DoS) vector where an attacker can call `connect()`, lock in a malicious treasury address, and cause

`AdminOperationalTreasury` initialization to fail irreversibly. This breaks the protocol setup flow and can brick strategy contracts before they are properly initialized.

Note: A similar issue exists in `addStrategy`, where invoking the strategy's `connect` function directly prevents admins from successfully adding a new strategy.

Recommendations

Restrict `connect()` access to only a trusted contract (e.g., via `onlyRole` or checking `msg.sender`).

[M-14] Reward Index Inflation Due to Rounding Up Can Trap Rewards

Severity

Impact: Medium

Likelihood: Medium

Description

In the `_depositPremium` function within `PositionInteractionFacet.sol`, when option premiums are added as rewards for liquidity providers, the `rewards.rewardsPerLPQ128` index is updated using `mulDivRoundingUp`:

```
if (lpSupply > 0) rewards.rewardsPerLPQ128 += upscaledPremium.mulDivRoundingUp
  (Q128, lpSupply);
```

Using `mulDivRoundingUp` causes the calculated reward added per LP token unit (scaled by `Q128`) to be rounded **up** to the nearest integer in the fixed-point representation. While seemingly small for individual deposits, these upward rounding errors accumulate in the `rewardsPerLPQ128` index with each premium deposit over time.

This leads to the `rewardsPerLPQ128` index potentially representing a slightly higher total reward distribution promise than the actual amount of `totalRewards` physically deposited into the pool.

When a user claims rewards via `_claimRewards` in `LiquidityFacet.sol`, their pending rewards (`pending`) are calculated based on the difference in the (potentially inflated) index relative to their last claim checkpoint. The function then attempts to subtract these `pending` rewards from the actual `totalRewards` available:

```
strg.ledger.state.rewardsByToken[token].totalRewards -= pending;
```

If the accumulated upward rounding errors in the `rewardsPerLPQ128` index cause the calculated `pending` amount for a user (especially one claiming later after many premium deposits have occurred) to exceed the actual remaining `totalRewards`, this subtraction will underflow and **revert the transaction**.

Consequently, the last users attempting to claim their rewards may be unable to do so, and their legitimately earned rewards become **permanently trapped** in the contract.

Recommendations

To prevent reward index inflation and ensure accounting solvency, use standard rounding down when calculating the reward-per-token index increment.

[M-15] Incorrect `ERC20` transfer in `withdrawFee` prevents fee withdrawal

Severity

Impact: Medium

Likelihood: Medium

Description

The `AdminOperationalTreasury.withdrawFee()` function uses `safeTransferFrom()` instead of `safeTransfer()` when withdrawing protocol fees. This is incorrect because:

1. `safeTransferFrom()` requires the spender to have an allowance from the owner.

2. The contract is trying to transfer from itself (`address(this)`) to the recipient.
3. The contract has not approved itself to spend its own tokens.

```
function withdrawFee(address token_, address recipient_) external {


    if (msg.sender != strg.setUp.base.feesReceiver) revert("FeesReceiverOnly");
    uint256 amount = strg.ledger.collectedFees[token_];
    strg.ledger.collectedFees[token_] = 0;

@>  IERC20Metadata(token_).safeTransferFrom(address(this), recipient_, amount);
    emit ProtocolFeeWithdrawn(token_, recipient_, amount);
}
```

For normal `ERC20`, `transferFrom` always checks the allowance of the spender, even when `from` is the same as `msg.sender`. As a result, the fee transfer will fail due to insufficient allowance.

Recommendation

Replace `safeTransferFrom()` with `safeTransfer()`, since the contract is transferring tokens it owns.

8.4. Low Findings

[L-01] Precision loss in fee calculation due to order of operations

In the `_calculateFeesUSD` function of `OperationalTreasury.sol`, the fee calculation is performed in a way that may lead to precision loss due to **Division Before Multiply**. The current implementation:

```
fees = price.mulDivRoundingUp(baseSetUp.protocolFees, FEES_UNIT);
fees = (fees * amount).toDecimals(36, 18);
```

The issue arises because the protocol fees are divided by `FEES_UNIT` before being multiplied by the `amount`. This order of operations can result in rounding down of intermediate values, **potentially leading to undercharging of fees.**

To minimize precision loss, the calculation should be reordered. The recommended implementation should be:

```
fees = (price * amount).mulDivRoundingUp(baseSetUp.protocolFees, FEES_UNIT);
fees = fees.toDecimals(36, 18);
```

[L-02] Missing `_disableInitializers()` in implementation contract

The `Oracle` contract inherits from `UUPSUpgradeable` and uses an `initializer` pattern, but it does not disable initializers in its constructor.

```
# The current implementation only has an init function:
function init(address owner) public initializer {
    _grantRole(AccessControlStorage.DEFAULT_ADMIN_ROLE, owner);
}
```

Without `_disableInitializers()` in the constructor, there's a risk that someone could initialize the implementation contract directly, which could

lead to unexpected behavior or potential security issues.

To mitigate this issue, it is recommended to add `_disableInitializers()` in the constructor of the implementation contract.

[L-03] Missing `IERC165` support in `supportsInterface()`

The `supportsInterface` function in the `ERC721WithURIBuilderUpgradeable` contract overrides the IERC165 interface but fails to provide support for the necessary interface IDs(`IERC165`). This is not a good practice and may return incorrect results.

```
function supportsInterface(bytes4 interfaceId)
        public
        view
        virtual
        override(IERC165, ERC721EnumerableUpgradeable)
        returns (bool)
    {
        return ERC721EnumerableUpgradeable.supportsInterface(interfaceId);
    }
```

To mitigate this issue, change it to:

```
function supportsInterface(bytes4 interfaceId)
        public
        view
        virtual
        override(IERC165, ERC721EnumerableUpgradeable)
        returns (bool)
    {
        return ERC721EnumerableUpgradeable.supportsInterface
          (interfaceId) || super.supportsInterface(interfaceId);
    }
```

[L-04] Changes in underlying token mid-operation may cause issues

The `setUnderlyingToken` function in the `AdminStrategy.sol` contract allows the underlying token to be changed at any time by any user with the appropriate role.

```
function setUnderlyingToken(address underlyingToken_) external onlyRole
      (AccessControlStorage.DEFAULT_ADMIN_ROLE) {
        StrategyStorage.layout().setUp.base.underlyingToken = underlyingToken_;
    }
```

This presents a certain risk, particularly if the `underlyingToken` is modified while operations dependent on it are in progress. For instance, if a strategy is executing trades or managing positions based on the current `underlyingToken`, changing the token midway can lead to inconsistent states and unexpected behavior.

To mitigate this issue, don't allow `setUnderlyingToken` after the strategy has been connected.

[L-05] Missing duplicate token validation in `setTargetWeight`

The `setTargetWeight` function lacks validation for duplicate tokens in the input array, which could lead to inconsistent weight assignments. Here's the problematic code:

```
function setTargetWeight(TokenWeight[] calldata tokens)
    external
    onlyRole(AccessControlStorage.DEFAULT_ADMIN_ROLE)
{
    PoolStorage.AssetsConfig storage assets = PoolStorage.layout().setUp.assets;
    uint256 nTokens = tokens.length;
    if (nTokens != assets.allAssets.length
      ()) revert PoolErrors.RequireAllTokens();

    uint256 total;
    TokenWeight memory item;
    uint256 weight;
    for (uint8 i; i < nTokens;) {
        item = tokens[i];
        assert(assets.allAssets.contains(item.token));
        weight = assets.isListed[item.token] ? item.weight : 0;
        assets.targetWeights[item.token] = weight;
        total += weight;
        // ... increment i
    }
    assets.totalWeight = total;
}
```

The issue arises because:

1. The function only checks if the number of tokens matches assets.allAssets.length().

2. There is no validation to ensure each token appears exactly once in the input array.
3. If a token appears multiple times in the input array:
    - Each occurrence will overwrite the previous weight assignment.
    - Only the last occurrence's weight will be stored.
    - The `totalWeight` calculation will include all occurrences.

To mitigate this issue, it is recommended to add validation to ensure each token appears exactly once(by checking address ordering).

[L-06] Role assignment mismatch in `PoolAdminFacet` initialization

Link: https://github.com/Hyperhyperfi/protocol/blob/212acfdcd4066334ff5ea06ce46b0bbca7ca212f/src/core/pool/facets/PoolAdminFacet.sol#L43-L53

There exists a critical issue in the initialization process of `PoolAdminFacet` where the initialization can fail due to a mismatch between role assignment and access control.

The problem occurs in the following sequence:

```
function init(
  addressowner,
  AddToken[]calldataassets,
  PoolStorage.BaseSetUpmemorybaseSetUp
)
    external
    initializer
{
    _grantRole(AccessControlStorage.DEFAULT_ADMIN_ROLE, owner);
    baseSetUp = _applyDefaultValues(baseSetUp);
    PoolStorage.layout().setUp.base = baseSetUp;
    addTokens(assets);  // This call will fail if owner != msg.sender
}

function addTokens(AddToken[] calldata newAssets) public onlyRole
  (AccessControlStorage.DEFAULT_ADMIN_ROLE) {
    // ... implementation
}
```

The issue arises because:

- The init function assigns the `DEFAULT_ADMIN_ROLE` to the specified `owner` address.
- Immediately after, it calls `addTokens` which has the `onlyRole(AccessControlStorage.DEFAULT_ADMIN_ROLE)` modifier.
- When `owner != msg.sender`, the initialization process fail because `msg.sender` doesn't have the required role when `addTokens` is called.

To mitigate this issue, modify the role assignment, and assign the `DEFAULT_ADMIN_ROLE` directly to `msg.sender`.

[L-07] Lingering permissions from uncleaned DEX swap token approvals

In `PositionInteractionFacet._swapAssetToAsset()`, the function approves `amountInMaximum` for swapping but the required `amountIn` could be less than or equal to `amountInMaximum`. This will leave the lingered approval to external Dex router.

```
function _swapAssetToAsset(
    ...
) internal returns (uint256 amountIn) {
    --- SNIPPED ---
@>    (uint256 amountInMaximum,,,) = IQuoterV2
  (dex.dexQuoter).quoteExactOutputSingle(quoteParams);
@>    amountInMaximum = _applySlippage(amountInMaximum);

@>    IERC20Metadata(assetIn).safeIncreaseAllowance
  (dex.dexRouter, amountInMaximum);


        tokenIn: assetIn,
        tokenOut: assetOut,
        fee: dex.dexFee,
        recipient: address(this),
        deadline: block.timestamp + 30 minutes,
        amountOut: amountOut,
        amountInMaximum: amountInMaximum,
        sqrtPriceLimitX96: sqrtPriceLimitX96
    });

@>  amountIn = IV3SwapRouter(dex.dexRouter).exactOutputSingle(params);

    --- SNIPPED ---
}
```

While this does not pose a significant security risk since:

1. Only the trusted router contract has the approval.

2. Future operations will reset approvals via `safeIncreaseAllowance`.

It is still best practice to clean up unused approvals to minimize the attack surface.

Consider clearing the approval after the swap.

[L-08] Dynamic `maxDepositPerUser` can stale Fenwick tree state

The protocol uses a Fenwick Tree to efficiently track user time-locked LP deposits, where each update and prefix sum operation is bounded by a configurable `maxDepositPerUser`.

However, the implementation assumes the tree is **consistently populated up to the current max**, which fails in two critical scenarios:

1. **When the max is increased after previously being lower**, the new nodes are **used in queries** (prefix sums) but were **never updated for the past deposits**.
2. **When the max is decreased** and if the max is increased again later, these **stale values corrupt prefix sums**.

This results in **incomplete or stale sums**, which breaks the accuracy of unlock logic and deposit tracking.

The Fenwick tree uses a binary indexing system where each node covers a specific range:

```
Node 1 (0001) = Sum of elements [1]
Node 2 (0010) = Sum of elements [1-2]
Node 3 (0011) = Sum of elements [3]
Node 4 (0100) = Sum of elements [1-4]
Node 5 (0101) = Sum of elements [5]
Node 6 (0110) = Sum of elements [5-6]
Node 7 (0111) = Sum of elements [7]
Node 8 (1000) = Sum of elements [1-8]
```

When `maxDepositPerUser` changes, consider the following scenarios:

1. Increase from 5 to 8:

- Node 8 should contain sum[1-8].
- But elements [1-5] never propagated up before.
- Result: Node 8 has an incomplete sum.

2. Decrease from 5 to 3 then back to 5:

- Nodes [4-5] retain stale values.
- New deposits don't properly update higher nodes.
- Result: Corrupted prefix sums.

However, this issue's practical impact is mitigated by the protocol's default configuration. The `maxDepositPerUser` is potentially set to a high value. With a max of high value, users would need to make deposits that fill most of these nodes. This makes the scenario unlikely in normal operations, though still possible if the protocol needs to adjust deposit limits frequently.

Consider ensuring that updating the `maxDepositPerUser` could lead to a potentially stale Fenwick tree state and ultimately lead to an incorrect unlock amount for users. To ensure prefix sums remain accurate when changing `maxDepositPerUser`, the protocol should implement a safe handling mechanism when resizing the Fenwick tree.

[L-09] Risk corruption from uncontrolled id in `setPositionsManager`

The `AdminOperationalTreasury.setPositionsManager()` function allows the protocol admin to replace the active `PositionsManager` contract. However, it does not enforce any validation on the new manager's position ID counter (`nextTokenId()`).

Position IDs are used as shared keys across multiple components, including position storage, strategy tracking, and payout logic. If a new `PositionsManager` is registered and begins minting IDs that **collide with existing ones**, it will overwrite or corrupt the existing position state, leading to data inconsistencies or even fund loss.

Consider adding a check when assigning a new position manager to ensure ID sequence continuity.

```
function setPositionsManager(address manager_) external onlyRole
  (AccessControlStorage.DEFAULT_ADMIN_ROLE) {
-     OperationalTreasuryStorage.layout().setUp.base.manager = IPositionsManager
- (manager_);
+     uint256 currentNextId = OperationalTreasuryStorage.layout
+ ().setUp.base.manager.nextTokenId();
+     if (currentNextId < IPositionsManager(manager_).nextTokenId
+ ()) revert InvalidPositionManager();
    OperationalTreasuryStorage.layout().setUp.base.manager = IPositionsManager
      (manager_);
}
```

[L-10] Uninitialized `maturityDate` in `Strategy.create()`

The `Strategy.create()` function declares a `maturityDate` return value but never initializes or sets it. This uninitialized value, which defaults to `0`, is returned and passed through to `OperationalTreasury._buy()`, where it's emitted via the `LiquidityLocked` event.

This leads to misleading or incorrect event logs, particularly showing `maturityDate = 0`.

The uninitialized value flows through to event emission:

```
function _buy(...) internal {
    // ...
    (sizeUSD, premium, maturityDate) = strategy.create(...);
    // maturityDate passed unchanged -> maturityDate: 0
    _emitEvent
      (optionID, amount, sizeUSD, buyToken, premium, fees, maturityDate);
}
```

Ensure that `Strategy.create()` explicitly sets and returns a `maturityDate`.

[L-11] Unbounded protocol fee issue in `setProtocolFees()`

The `setProtocolFees` function in `AdminOperationalTreasury` allows the admin to set protocol fees without any maximum cap or constraints:

```
function setProtocolFees(uint24 protocolFees_) external onlyRole
      (AccessControlStorage.DEFAULT_ADMIN_ROLE) {


        emit ProtocolFeesSet(baseSetUp.protocolFees, protocolFees_);
        baseSetUp.protocolFees = protocolFees_;
    }
```

This introduces two vulnerabilities:

1. The admin can set the fee to 100% (or more), enabling the protocol to transfer the entire user allowance when calling `_takeFeesLockLiquidity` to transfer fee+premium from the user buying the option.
2. Similarly, during `payOff`, the admin could configure the fee to absorb all user PnL as protocol revenue.

[L-12] Oracle: irreversible token configuration

The `configToken` function allows admins to whitelist a token and set its price feed.

```
function configToken(address token, address priceFeed) external onlyRole
      (AccessControlStorage.DEFAULT_ADMIN_ROLE) {
        if (address(tokenOracle[token]) != address
          (0)) revert TokenAlreadyConfigured();
        else whitelistedTokens.push(token);

        tokenOracle[token] = AggregatorV3Interface(priceFeed);
        emit TokenAdded(token);
    }
```

However, once a token is configured, it cannot be updated or removed. This becomes problematic if the token or its associated price feed becomes faulty, deprecated, or incorrect.

There is no way to recover from misconfiguration or respond to issues with oracles, which introduces operational risk and limits the flexibility of the system.

[L-13] PoolAdminFacet: `removeToken()` lacks protocol fee check

The `removeToken` function does not check if a token has a non-zero protocol fee before removal. This can prevent the feeDistributor from claiming fees after the token is removed.

```
function removeToken(address _token) external onlyRole
  (AccessControlStorage.DEFAULT_ADMIN_ROLE) {
    PoolStorage.Layout storage strg = PoolStorage.layout();
    if
      (strg.ledger.state.poolAmount[_token] != 0) revert PoolErrors.PoolNotEmpty(_toke
    strg.setUp.assets.allAssets.remove(_token);
    strg.setUp.assets.stablecoins.remove(_token);

    if
      (strg.ledger.state.rewardsByToken[_token].totalRewards == 0) strg.setUp.assets.r
}
```

[L-14] Strategy cannot be added after removal in AdminOperationalTreasury

Admin Can not add already removed strategy, because `addStrategy` function calls `connect` in strategy but it will revert since can be called once:

```
function addStrategy(IStrategy s) external override onlyRole
      (AccessControlStorage.DEFAULT_ADMIN_ROLE) {


        if (setUp.acceptedStrategy[s]) revert StrategyAlreadyAdded();
        _connect(s, setUp);
    }

    function _connect
      (IStrategy s, OperationalTreasuryStorage.SetUp storage strg) internal {
        s.connect();
        strg.acceptedStrategy[s] = true;
    }
```

`connect` function in `AdminStrategy` contract can be called only once:

```
function connect() external override {
        StrategyStorage.SetUp storage setUp = StrategyStorage.layout().setUp;
        IOperationalTreasury treasury_ = IOperationalTreasury(msg.sender);
        if (address(setUp.treasury) != address(0)) revert TreasuryAlreadySet();
        setUp.treasury = treasury_;
    }
```

[L-15] `payOff()` blocked for option exercise when contract is paused

The `OperationalTreasury.payOff` function, used by option holders to exercise their positions after maturity, internally calls the `exerciseOrLiquidatePosition` function on the `PositionInteractionFacet.sol` contract. The `exerciseOrLiquidatePosition` function includes the `whenNotPaused` modifier.

Consequently, if the Pool contract is paused (e.g., by an admin for emergency maintenance, upgrade, or due to external dependency issues), users cannot call `payOff` to exercise their options. If an option is In-The-Money (ITM) during the pause but becomes Out-of-The-Money (OTM) by the time the contract is unpaused, the user **loses the opportunity to realize their profit**. This directly impacts the financial outcome for the option holder based on administrative action (pausing).

Explicitly document this behavior and the associated risk for option holders in the protocol's user-facing documentation. Users should be aware that the ability to exercise options might be temporarily suspended if the contract is paused.

[L-16] Inefficient payout transfer check in `payOff()`

The `payOff` function of `OperationalTreasury.sol` calculates the net profit (`netAmount`) by subtracting fees (`feesUSD`) from the gross profit (`pnl`). However, the final transfer of this net profit to the user is conditional on the gross profit (`pnl`) being positive, rather than the net amount (`netAmount`) itself:

```
uint256 netAmount = pnl - feesUSD;
// ...
if (pnl > 0) baseSetUp.token.safeTransfer(account, netAmount);
```

This logic is inefficient because even if the gross profit (`pnl`) is positive, the net amount (`netAmount = pnl - feesUSD`) can be zero if the calculated fees equal the profit. In such a case where `pnl > 0` but `netAmount == 0`, the condition `if (pnl > 0)` still evaluates to true, leading to an unnecessary

`safeTransfer` call attempting to transfer zero. Attempting a zero-value transfer wastes gas without changing any state.

Modify the condition to check if `netAmount` is strictly greater than zero before attempting the transfer.

[L-17] Missing initializer guard in `_init()` function

The `internal` function `_init` within `ERC721WithURIBuilderUpgradeable.sol` lacks the `onlyInitializing` modifier. This function is called by the `initializer` function (`init`) of the upgradeable implementation contract `PositionsManager.sol`.

While the main `PositionsManager.init` function *is* protected by the `initializer` modifier, and the underlying `__ERC721_init` called by `_init` also has protection, the absence of the guard on `_init` itself poses a latent risk. If future upgrades to `PositionsManager` were to introduce new functions that mistakenly call `_init` again after initial deployment, there would be no direct protection at the `_init` level to prevent this call attempt.

Recommendation:

Add the `onlyInitializing` modifier to the `ERC721WithURIBuilderUpgradeable._init` function to explicitly enforce its initialization-only context and adhere to standard upgradeable contract patterns.

```
function _init
      (string memory name_, string memory symbol_) internal onlyInitializing {
        __ERC721_init(name_, symbol_);
    }
```

[L-18] Contract wallets incompatible with `EIP-1271` signature verification

The signature verification mechanism in `IvSignerRecovery.sol` relies exclusively on ECDSA recovery for validating signatures:

```
function _recoverSigner(
    SplitSig memory sig,
    uint256 iv,
    IStrategy strategy,
    uint256 maturityDate,
    uint256 relativeStrike
) internal view returns (address) {
    // Uses ECDSA recovery directly
    return _digest(iv, strategy, maturityDate, relativeStrike).recover
      (sig.v, sig.r, sig.s);
}
```

This implementation only supports signatures from Externally Owned Accounts (EOAs) and lacks compatibility with **EIP-1271** (`Standard Signature Validation Method for Contracts`). As a result, users with smart contract wallets (such as Gnosis Safe) cannot use the protocol's option buying functionality in the `OperationalTreasury.buy()` function, because their contract-based signatures cannot be verified by `ECDSA.recover`.

With the growing adoption of smart contract wallets for enhanced security features and multi-signature capabilities, this limitation unnecessarily restricts the protocol's user base.

Extend the signature verification logic to support EIP-1271. This typically involves checking if the signer's address corresponds to a contract. If it is a contract, call its `isValidSignature` function to confirm the signature's validity according to the contract's own logic.

[L-19] `_computeMaturityDate` `uint32` timestamp limits protocol lifespan

The `_computeMaturityDate` function in `PositionParams.sol` casts maturity timestamps to `uint32` before returning them:

```
function _computeMaturityDate(uint256 period) internal view returns
      (uint32) {
        uint256 maturityTimestamp = block.timestamp + period;
        maturityTimestamp -= (maturityTimestamp % 1 days);
        maturityTimestamp += 1 days - 1;
        return uint32(maturityTimestamp); // <-- Cast to uint32
    }
```

This imposes a hard limit on the maximum representable timestamp. A `uint32` can only represent Unix timestamps up to **4,294,967,295**, which corresponds to **February 7, 2106**. After this date, timestamps calculated using

`block.timestamp` will overflow when cast to `uint32`, resulting in incorrect maturity dates.

This function is used in core protocol components, including option creation (`Strategy::_create`) and signature verification (`OperationalTreasury::_checksOnBuy`).

While this limitation is far in the future (approximately 81 years from the current date in April 2025), it represents a **hard end-of-life** for the protocol in its current form unless addressed. For a financial protocol potentially designed for long-term operation, this creates an unnecessary and avoidable time constraint.

Replace the use of `uint32` for maturity timestamps throughout the protocol with a larger unsigned integer type.

[L-20] Missing parent initializer call in `__UUPSUpgradeable_init()`

The `Oracle.sol` contract inherits from `UUPSUpgradeable` but its `init` function fails to call the corresponding parent initializer function, `__UUPSUpgradeable_init()`. While this parent initializer function might be empty in the current version of the OpenZeppelin library used, omitting the call violates the explicit initialization pattern required by OpenZeppelin Upgradeable contracts. This deviation poses risks for forward compatibility should the library be updated with logic in that initializer, and makes the code non-standard and harder to maintain.

Adhere to the standard OpenZeppelin Upgradeable pattern by explicitly calling `__UUPSUpgradeable_init()` within the `Oracle.init()` function.

```
function init(address owner) public initializer {
        __UUPSUpgradeable_init(); // Initialize UUPSUpgradeable parent
        _grantRole(AccessControlStorage.DEFAULT_ADMIN_ROLE, owner);
    }
```

[L-21] Missing slippage protection in strike price calculation for options

The protocol's option creation process lacks protection against price slippage between the time a user submits a transaction and when it's executed. Here's the problematic flow:

```
// In OperationalTreasury.sol
function buy(
    IStrategy strategy,
    address holder,
    uint256 amount,
    uint256 period,
    uint256 relativeStrike,  // User inputs relative strike
    uint256 iv,
    IvSignerRecovery.SplitSig memory sig
) external virtual override nonReentrant whenNotPaused {
    _checksOnBuy(strategy, period, relativeStrike, iv, sig);
    _buy(holder, strategy, amount, period, relativeStrike, iv);
}

// In Call.sol
function _calculateStrike(uint256 preciseCurrentPrice, uint256 relativeStrike)
    internal
    view
    override
    returns (uint256)
{
    uint256 calculatedStrike = preciseCurrentPrice + (
      (preciseCurrentPrice * relativeStrike) / 100);
    return _roundPrice(calculatedStrike);
}

// In Put.sol
function _calculateStrike(uint256 preciseCurrentPrice, uint256 relativeStrike)
    internal
    view
    override
    returns (uint256)
{
    uint256 calculatedStrike = preciseCurrentPrice - (
      (preciseCurrentPrice * relativeStrike) / 100);
    return _roundPrice(calculatedStrike);
}
```

The issue arises because:

1. Users specify a `relativeStrike` percentage for their desired strike price.
2. The actual strike price is calculated using the `currentPrice` at execution time.
3. Between transaction submission and execution:
    - The underlying asset price may change significantly.
    - This leads to a different strike price than what the user intended and users have no control over the final strike price.

For example:

1. User wants a 10% call when ETH is $2000.

2. Expected strike: $2200.
3. If the price moves to $2100 before execution.
4. Actual strike: $2310 (10% above $2100).

This could make the option much more expensive or less profitable, and this could lead to a bad user experience.

Recommendations:

Implement slippage protection for strike price calculation.

[L-22] Omission of `_refreshVirtualPoolValue()` causes stale fees and PLP price

The `_payout()` function in `PositionInteractionFacet` updates pool balances but fails to update the virtual pool state and emit updated PLP pricing. This leads to stale virtual pool values used in dynamic fee calculations and outdated PLP price reporting.

The issue occurs because:

1. Pool balance changes in `_payout()`: `poolAmount[pos.uAsset]` and/or `poolAmount[pos.buyToken]`, affect total pool value.

2. Stale `virtualNetPoolValue` values affect dynamic fee application:

```
function _calcFeeRate(...) internal view returns
  (int256 delta, uint256 feeRate) {
    if (PoolStorage.layout().setUp.assets.totalWeight == 0
            || PoolStorage.layout().ledger.state.virtualNetPoolValue == 0
    ) return (0, feeRate);
}
```

3. PLP price emissions are also missed.

Recommendation:

Apply `_refreshVirtualPoolValue()` and `_emitPLPPrice()` called in `_payout()`.

[L-23] Token mismatch in `operationalTreasury` affects pool payouts

The protocol potentially has inconsistent token transfer for `pnl` payouts between `OperationalTreasury` and `Pool` contracts. While `Pool` uses the position's `buyToken` for payouts, `OperationalTreasury` uses the current `baseSetUp.token`. If `OperationalTreasury` is updated and reinitialized the base setup with a different base token, users will receive payouts in the wrong token.

```
function payOff(
  uint256positionID,
  addressaccount
) external override nonReentrant whenNotPaused {
    --- SNIPPED ---
    if (pnl > 0) baseSetUp.token.safeTransfer(account, netAmount);
    emit Paid(positionID, account, netAmount, feesUSD);
}
```

```
function _payout(
  Positionmemorypos,
  PositionClosememoryclose
) internal returns (uint256 pnl
    --- SNIPPED ---
    strg.ledger.state.poolAmount[pos.buyToken] -= pnl;
    _doTransferOut(pos.buyToken, msg.sender /* operational treasury*/, pnl);
}
```

Recommendation:

Record the original `baseSetUp.token` or `buyToken` with each position in the `OperationalTreasury` and use it for payouts.

[L-24] `updateDexConfig()` in `poolAdminFacet` ignores pair-specific DEX fees

The `updateDexConfig()` function in `PoolAdminFacet` sets a single fee value per DEX router address:

```
function updateDexConfig(address router, uint24 fee, bool isValid)
    external
    onlyRole(AccessControlStorage.DEFAULT_ADMIN_ROLE)
{
    PoolStorage.Dex storage dex = PoolStorage.layout().setUp.dex;
    dex.valid[router] = isValid;
@>  dex.fee[router] = fee;
    emit DexUpdated(router, isValid, fee);
}
```

However, this structure fails to accommodate DEXes like Uniswap V3, which supports multiple pools per token pair with different fees (e.g., 500, 3000, 10000). Using a single fee for all swaps on a given router can result in a revert or higher slippage in `_swapAssetToAsset` function.

Recommendations:

Update the design to allow pair-specific fee configuration.

[L-25] Inconsistent decimal handling in liquidity calculation

The `_calcAddLiquidity` function inconsistently handles decimal places during liquidity calculations:

1. **Initial Calculation (Correct):**
   - Uses `provDecimals` (from price feed) when calculating `valueChange`:

```
(tokenPrice, provDecimals) = _getLastPrice(_token);
valueChange = (_amountIn * tokenPrice).toDecimals
  (tokenDecimals + provDecimals, 18);
```

2. **LP Amount Calculation (Incorrect):**
   - Hardcodes 8 decimals instead of using `provDecimals`:

```
lpAmount = lpAmount.toDecimals
//(tokenDecimals + 8, plpDecimals);  // Should use provDecimals
```

This inconsistency occurs in both branches of the LP amount calculation (when `lpSupply == 0` and the else case). The function assumes price feed decimals are always 8, which may not be true for all oracle providers.

Recommendations

Use `proveDecimals` consistently throughout the codebase to standardize handling of different token price decimal formats.

[L-26] End-of-day option buys vulnerable due to `block.timestamp` dependency

The `buy()` function in `OperationalTreasury.sol` allows users to purchase options by providing parameters including a signature (`sig`) generated off-chain by a trusted signer. This signature covers several parameters, including a `maturityDate` which is derived from the requested `period`.

The verification process happens in the internal `_checksOnBuy` function:

```
function _checksOnBuy(
        IStrategy strategy,
        uint256 period,
        uint256 relativeStrike,
        uint256 iv,
        IvSignerRecovery.SplitSig memory sig
    ) internal view {
        // ... other checks ...
        // maturityDate is recalculated here using block.timestamp
        address recoveredSigner = _recoverSigner
          (sig, iv, strategy, period._computeMaturityDate(), relativeStrike);
        if (recoveredSigner != strg.setUp.base.signer) revert InvalidIVSigner();
    }
```

The issue lies in how the `maturityDate` used for signature verification is calculated on-chain via `period._computeMaturityDate()`. This function, defined in `PositionParams.sol`, calculates the maturity timestamp based on the **current** `block.timestamp` at the time of execution:

```
function _computeMaturityDate(uint256 period) internal view returns
      (uint32) {
        uint256 maturityTimestamp = block.timestamp + period; // Uses current
        // block.timestamp
        maturityTimestamp -=
        //(maturityTimestamp % 1 days); // Rounds down to start of day
        maturityTimestamp += 1 days - 1; // Sets to 23:59:59 of that day
        return uint32(maturityTimestamp);
    }
```

Because `_computeMaturityDate` depends on `block.timestamp` and rounds the result, the following scenario occurs:

1. The off-chain signer service calculates `maturityDate` based on its current timestamp (e.g., Day 1, 23:55:00) and signs the payload including this date (e.g., resulting in Day X, 23:59:59).
2. A user submits the `buy()` transaction with this signature near the end of Day 1.
3. Due to network congestion or block timing, the transaction is included in a block mined *after* midnight (e.g., Day 2, 00:01:00).
4. During execution, `_checksOnBuy` calls `period._computeMaturityDate()`. This recalculates the `maturityDate` using the `block.timestamp` from Day 2. The result will be exactly 24 hours later than the `maturityDate` originally signed (e.g., Day X+1, 23:59:59).
5. The `_recoverSigner` function attempts to verify the signature against a data payload containing this **new, later** `maturityDate`.
6. The signature verification fails because the signed data (with the earlier `maturityDate`) does not match the data recalculated on-chain (with the later `maturityDate`). The transaction reverts with `InvalidIVSigner`.

This effectively creates a **Denial of Service (DoS)** condition for users attempting to buy options near the end of a UTC day, especially during times of network congestion. Users will experience failed transactions and loss of gas fees because the `maturityDate` used for verification shifts based on `block.timestamp` between signing and execution.

Recommendations:

To resolve this, the `maturityDate` used for signature verification must be consistent between the time of signing and the time of execution. The dependency on `block.timestamp` during verification for signed data should be removed.

1. **Sign the Explicit Maturity Date:** The off-chain signer should calculate the exact `maturityDate` timestamp (e.g., using the current logic or a fixed daily calculation) and include this **explicit timestamp** in the data payload that gets signed.
2. **Pass Explicit Date to `buy()`:** Modify the `buy` function signature to accept this explicit `maturityDate` timestamp that was signed.
3. **Use Explicit Date in Verification:** Modify `_checksOnBuy` to pass this **explicit** `maturityDate` (received as an argument) to `_recoverSigner` instead of recalculating it using `period._computeMaturityDate()`.

[L-27] Token removal can be blocked by minimal deposits

In the current implementation, the protocol's token removal functionality can be permanently blocked by a malicious actor through a minimal token deposit. This vulnerability exists due to the strict validation in PoolAdminFacet.sol:

```
function removeToken(address _token) external onlyRole
      (AccessControlStorage.DEFAULT_ADMIN_ROLE) {
        PoolStorage.Layout storage strg = PoolStorage.layout();
        if
          (strg.ledger.state.poolAmount[_token] != 0) revert PoolErrors.PoolNotEmpty(_
        strg.setUp.assets.allAssets.remove(_token);
        strg.setUp.assets.stablecoins.remove(_token);

        if
          (strg.ledger.state.rewardsByToken[_token].totalRewards == 0) strg.setUp.asse
    }
```

Any user can deposit a minimal amount (as small as 1 `wei`) of a token into the pool. Once a token has any non-zero balance, the admin cannot remove it from the protocol. This creates a permanent denial-of-service (DOS) vector for token removal functionality.

This issue is more severe if one of the stable coins depends. Being Unable to remove the coin from the `strg.setUp.assets.stablecoins` could cause an incorrect `getPoolValue` calculation.

Recommendations:

Implement a more robust token removal system that allows administrators to remove tokens even with minimal balance.

[L-28] Incorrect decimal conversion in `LiquidityFaucet` calculation

In the `_calcAddLiquidity` function, the `lpAmount` conversion uses an incorrect decimal calculation:

```
function _calcAddLiquidity(address _token, uint256 _amountIn)
        internal
        view
        returns (
          uint256netAmount,
          uint256pFee,
          uint256lpAmount,
          uint256tokenPrice
        )
    {
        --snip--
        if (lpSupply == 0) {
            lpAmount = netAmount * tokenPrice;
            lpAmount = lpAmount.toDecimals(tokenDecimals + 8, plpDecimals);
        } else {
@1>          lpAmount = netAmount.mulDiv(tokenPrice * lpSupply, _getPoolValue
  (false, false));
@2>          lpAmount = lpAmount.toDecimals(tokenDecimals + 8, plpDecimals);
        }
    }
```

At step `@1>`, the correct way to calculate the `lpAmount` decimals should be:

```
lpAmount decimals =
  (tokenDecimals * priceDecimals * plpDecimals) / poolValueDecimals(18)
```

However, at step `@2>`, the code incorrectly assumes that the decimals are `tokenDecimals + 8` and attempts to convert it to `plpDecimals`. On the other hand, the code assumes that `plpDecimals` equals `poolValueDecimals (18)`.

Using this incorrect formula for decimal conversion can lead to inaccurate LP token allocation, which directly impacts user shares and the overall integrity of the pool's accounting.

Recommendations:

Update the `lpAmount` decimals conversion logic to use the correct decimal calculation.

[L-29] Incorrect `EIP-712` data hashing implementation

The `IvSignerRecovery.sol` contract aims to use EIP-712 for verifying signatures related to option parameters (`iv`, `strategy`, `maturityDate`, `relativeStrike`) needed in the `OperationalTreasury.buy()` function. However, the implementation of the structured data hashing within the `_hash` function deviates significantly from the EIP-712 standard.

The EIP-712 standard specifies that the hash of a structure (`structHash`) should be calculated as: `hashStruct(s : 𝕊) = keccak256(typeHash ‖ encodeData(s))` Where:

- `typeHash` is the `keccak256` hash of the structure's type definition string.
- `encodeData(s)` ABI-encodes the struct members, crucially encoding dynamic types like `string` and `bytes` as the `keccak256` hash of their content.

The current `_hash` function calculates the hash as follows:

```
function _hash(
      uint256iv,
      IStrategystrategy,
      uint256maturityDate,
      uint256relativeStrike
    )
        internal
        view
        returns (bytes32)
    {
        (string memory uAsset, string memory oType) = _getAssetAndType
          (strategy);
        // Incorrect hashing: Missing typeHash and incorrect string encoding
        return keccak256(abi.encode
          (iv, uAsset, oType, maturityDate, relativeStrike));
    }
```

This implementation has two critical flaws regarding the EIP-712 standard:

1. **Missing `typeHash`:** It completely omits the `typeHash` component. The `typeHash` prevents signature collisions between different data structures.
2. **Incorrect Encoding of Dynamic Types:** It uses `abi.encode` for the `string` variables `uAsset` and `oType`. EIP-712 requires dynamic types (`string`, `bytes`) within `encodeData` to be represented by the `keccak256` hash of their byte content (i.e., `keccak256(bytes(uAsset))` and `keccak256(bytes(oType))`).

- **High Risk of Signature Mismatch (DoS):** If the off-chain signing service **correctly** implements the EIP-712 standard (calculating `typeHash` and hashing strings properly), the signatures generated off-chain will **never** match the hash calculated by the non-standard `_hash` function on-chain. This will cause all `buy()` transactions to fail signature verification, resulting in a Denial of Service for the core functionality of purchasing options.

Recommendations:

Refactor the `_hash` function to strictly adhere to the EIP-712 `hashStruct` specification.

1. **Define the `TYPEHASH`:** Define the type string for the signed data structure and compute its `keccak256` hash. This can be stored as a constant.

2. **Implement Correct `encodeData`:** Modify the hashing logic to prepend the `SIGNED_DATA_TYPEHASH` and correctly encode the members according to EIP-712 rules (hashing dynamic types like `string`).

[L-30] Option buyers choose volatility to reduce premium

The implied volatility plays a crucial role in the Black-Scholes equation to calculate the option prices:

For a **European call option**:

$C = S_0 \times N(d_1) - K \times e^{\wedge}(-rT) \times N(d_2)$

For a **European put option**:

$P = K \times e^{\wedge}(-rT) \times N(-d_2) - S_0 \times N(-d_1)$

Where:

- (C) = Call option price
- (P) = Put option price
- (S_0) = Current price of the underlying asset
- (K) = Strike price
- (r) = Risk-free interest rate (annualized)
- (T) = Time to maturity (in years)
- (σ) = Volatility of the underlying asset (annualized)
- (N(x)) = Cumulative distribution function (CDF) of the standard normal distribution

Also, the coefficients (d_1) and (d_2):

$d_1 = [\ln(S_0 / K) + (r + (½) \times \sigma^2) \times T] / (\sigma \times \sqrt{T})$,
$d_2 = d_1 - \sigma \times \sqrt{T}$

As it is clear from the abovementioned formula, if we decrease the σ (volatility), the d_1 and d_2 increase respectively. However, the key point here is that even if N(x) increases, the strike $N(d_2)$ increases more than the spot $N(d_1)$, and the overall premium decreases.

Also, in the case where $N(d_2)$ becomes more than $N(d_1)$, the contract makes the premium become zero:

```
call = strikeNd2 <= spotNd1 ? spotNd1 - strikeNd2 : 0
```

[L-31] High gas use risk from `Uniswap V3` quoter

The `PositionInteractionFacet` contract uses Uniswap V3's quoter contract for on-chain price calculations during position exercises and liquidations. As noted in Uniswap's documentation, these quoter functions are not designed for on-chain use due to their high gas consumption.

The issue is present in the `_swapAssetToAsset` function:

```
function _swapAssetToAsset(
    address assetIn,
    address assetOut,
    uint256 amountOut
) internal returns (uint256 amountIn) {
    PoolStorage.Layout storage strg = PoolStorage.layout();
    PoolStorage.Dex memory dex = strg.setUp.base.dex;

    // Gas-intensive quote operation

        tokenIn: assetIn,
        tokenOut: assetOut,
        amount: amountOut,
        fee: dex.dexFee,
        sqrtPriceLimitX96: 0
    });
    (uint256 amountInMaximum,,,) = IQuoterV2
      (dex.dexQuoter).quoteExactOutputSingle(quoteParams);

    // Swap execution

        tokenIn: assetIn,
        tokenOut: assetOut,
        fee: dex.dexFee,
        recipient: address(this),
        deadline: block.timestamp + 30 minutes,
        amountOut: amountOut,
        amountInMaximum: amountInMaximum,
        sqrtPriceLimitX96: 0
    });

    amountIn = IV3SwapRouter(dex.dexRouter).exactOutputSingle(params);
}
```

This implementation is problematic for several reasons:

1. The quoter functions simulate the entire swap path to calculate prices, making them extremely gas-intensive.
2. These functions are explicitly not designed for on-chain use according to Uniswap's documentation.
3. The quotes are used in critical position management functions (exercise/liquidation).

The risk is particularly high because position exercises/liquidations are most likely to occur during volatile market conditions, which are exactly the conditions where gas costs are highest.

Recommendations:

1. Use an oracle-based approach for price calculations.
2. Alternatively, implement optimistic execution with reasonable bounds.

[L-32] `Indexed` keyword in events causes struct data loss

When the `indexed` keyword is used for reference type variables such as dynamic arrays or structs, it will return the hash of the mentioned variables. Thus, the event, which is supposed to inform all of the applications subscribed to its emitting transaction (e.g. front-end of the DApp, or the backend listeners to that event), would get a meaningless and obscure 32 bytes that correspond to keccak256 of an encoded struct/array. This may cause some problems on the DApp side and even lead to data loss.

The problem exists inside the `IStrategy` contract. The event `Acquired` is defined in such a way that the data field, which is a struct instance, is indexed. With doing so, the expected parameter wouldn't be emitted properly and the front-end would get meaningless one-way hashes.

We can see the effect of omitting the `indexed` keyword for the event by defining another event named `Acquired1`:

```
event Acquired(
        uint256 indexed id,
        StrategyData indexed data,
        uint256 sizeUSD
    );

    event Acquired1(
        uint256 indexed id,
        StrategyData data,
        uint256 sizeUSD
    );

    struct StrategyData {
        uint128 amount; // 18 decimals
        uint128 strike; // 18 decimals
    }

    function eventEmitter() public {
        StrategyData memory st1 = StrategyData({amount: 42e6, strike: 5});
        emit Acquired(1, st1, 100);
        emit Acquired1(1, st1, 100);
    }
```

The result would be:

event `Acquired`:

```
{
                "from": "0x5Ea69b5B2f499Ad9f1127E77d1179cCb471b5e19",
                "topic": "0x8139f37233b28b6f84e65bdca1901a93aaf3c54ef5572840702eb6d3a3
                "event": "Acquired",
                "args": {
                        "0": "1",
                        "1": {
                                "_isIndexed": true,
                                "hash": "0xf4941371d1251c4c97c60a36bd52b60338359139294
                        },
                        "2": "100"
                }
```


event `Acquired1`:


```
{
                "from": "0x5Ea69b5B2f499Ad9f1127E77d1179cCb471b5e19",
                "topic": "0x303c7cf269a5457a46482b73a49633deaacadb51f603cb724dceebbcb6
                "event": "Acquired1",
                "args": {
                        "0": "1",
                        "1": [
                                "42000000",
                                "5"
                        ],
                        "2": "100",
                        "id": "1",
                        "data": [
                                "42000000",
                                "5"
                        ],
                        "sizeUSD": "100"
                }
        }
```


Recommendation:

```
event Acquired(
        uint256 indexed id,
-       StrategyData indexed data,
+       StrategyData data,
        uint256 sizeUSD,
        uint256 premium,
        uint256 period,
        uint256 indexed iv
    );
```