

Pashov Audit Group

# OLY
# Security Review

December 31st 2025 - January 20th 2026

# Contents

1. About Pashov Audit Group 4
2. Disclaimer 4
3. Risk Classification 4
4. Methodology 5
5. About OLY 5
6. Executive Summary 5
7. Findings 7

Critical findings 11

[C-01] Genesis rewards will be stolen 11
[C-02] Underflow during `FarmKeeper` V4 liquidity minting 12
[C-03] V4 farm tokenId can be cached but not minted 14
[C-04] Intermediary `poolId` not updated for nonnative farms 16
[C-05] Incorrect handling of `stETH` shares causes accounting error in `ValidatorVa` 19

High findings 21

[H-01] Fail to unwrap WETH in `swapV3EthForFarmToken` 21
[H-02] Unused amount may be calculated incorrectly 22
[H-03] Limit order incorrectly filled on tick movement 23
[H-04] Inefficient liquidity provision due to decimal-agnostic token balance comparison 26
[H-05] First depositor can claim all auction tokens before auction ends 27
[H-06] Liquidity addition vulnerable to sandwich attacks due to spot price manipulation 28
[H-07] Asymmetric unit scaling in `swapAmount` calculation breaks rebalancing logic 29
[H-08] Exact output swaps incur lower tax than exact input swaps due to tax calculation 31
[H-09] Withdraw pays only one token while accounting debits both in filled epochs 32

Medium findings 35

[M-01] Reward loss due to precision underflow in cycle payouts 35
[M-02] Share rate increase causing staking value loss 36
[M-03] Reward claim pattern issue with blacklisted tokens 37
[M-04] Eth pricing fallback misprices USD value 38
[M-05] Overflow risk in price calculation during V4 pool creation 40
[M-06] Inflation pool accounting gap allows excess distribution 41
[M-07] Tokens permanently locked in accounting if auction receives no deposit 43
[M-08] Accrued fees can be exploited to obtain subsidized liquidity positions 44
[M-09] New limit order participants unfairly receive fees from exited positions 45
[M-10] Withdrawal requests revert when combined amounts exceed Lido's `MAX_WITHDRAWAL` 47
[M-11] Imprecise maturity calculation allows premature stake ending 49
[M-12] Incorrect validation in `validateFarmSetup` allows non-reward tokens in V4 farms 50
[M-13] Missing validation allows intermediary pool with wrong token causing fund loss 51
[M-14] Reward debt heuristic can drop rewards after reward token readdition 52
[M-15] Excess ETH not refunded when placing limit orders 53
[M-16] Unequal reward distribution lets short-term stakers access long-term rewards 54
[M-17] Inflation reward calculation is not accurate 56
[M-18] Ethereum can be locked if required liquidity is not achieved 57
[M-19] Emergency withdrawal fails to recover ETH in `pendingEth` leaving funds in `V` 58
[M-20] Removed tokens lead to unclaimable rewards causing permanent loss of user funds 59

[M-21] Missing slippage protection in `mint` function exposes users to token loss 60
[M-22] Dead stake violates zero staker invariant and absorbs inflation share 62
[M-23] LP fees can be permanently locked in nonfilled epochs after all users call 63
[M-24] Missing collect accrued swap fees in `unlockCallbackPlace` 64
[M-25] Protocol rebalances incur `OLY` sell tax when swapping via taxed v4 farm pool 65
Low findings 67
[L-01] Stake position flooding risk 67
[L-02] Ineffective staking allowance management in `LaunchManager` due to hardcoding 68
[L-03] Claimable amount underflow causing stuck token 68
[L-04] `EthPrice` oracle missing price deviation check 69
[L-05] Mint transactions vulnerable to griefing via front running due to no partials 70
[L-06] `ValidatorVault.harvest()` reverts on buy and burn rewards below Lido minimum 70
[L-07] `StakeETH` allows ETH to be staked while minting zero `wstETH` 71
[L-08] Unused token allowance after partial swap execution 71
[L-09] Payout schedule advances only one cycle per call when cycles are overdue 71
[L-10] `CyclePayoutTriggered` logs `ethReward = 0` due to reward token ordering issue 72
[L-11] No `wstETH/stETH` oracle on ETH mainnet 73
[L-12] Self-referral allows users to claim both minter and referrer bonuses 74
[L-13] Farm removal fee distribution mismatch with documentation 75
[L-14] Incorrect liquidity removed value in `FarmKeeper_Removed` event 76
[L-15] Constructor does not validate price configuration parameters 76
[L-16] Intermediary pool configuration lacks validation 77
[L-17] Twap period can be set to an invalid value 77
[L-18] Reward tokens become permanently locked when removed with no active share 77
[L-19] Rounding dust from reward distribution becomes permanently locked 78
[L-20] Summed reward claims across different tokens produce meaningless values 78
[L-21] Inconsistent slippage handling for increase and decrease of liquidity in v4 l 79
[L-22] Implicit assumption on Chainlink price feed decimals in `EthPrice` 79
[L-23] `StakeETH` can be front run with a small amount to disrupt user transactions 80
[L-24] Missed slippage protection when users sell OLY 80
[L-25] Mint accounting break with mixed liquidity and vesting configuration 81
[L-26] Insufficient observation cardinality validation in `FarmKeeper.setTwapPeriod` 82
[L-27] ValidatorVault assumes `stETH` is pegged 1 to 1 with `ETH` 83

# 1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out 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.

# 3. Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

**Impact**

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

**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

## 4. Methodology

Pashov Audit Group conducts each engagement as a team that works through the codebase end-to-end: we map the system, hunt for bugs, and then collaborate with the client to remediate the findings. This report tracks every commit reviewed, every finding raised, and the resolution status of each issue. Our auditors draw on experience from private engagements and public contests, and pick their own toolset per project — static analysis, fuzzers, formal verification, and AI-assisted review — based on what fits the codebase.

## 5. About OLY

OLY is a comprehensive token launch and DeFi yield platform that combines token minting with automatic staking, Uniswap V3/V4 liquidity farming, and Lido ETH staking across multiple reward cycles. The system features a governance-enabled staking vault with 5 different payout schedules, treasury-managed liquidity positions, and automated revenue distribution mechanisms.

## 6. Executive Summary

A time-boxed security review of the **De-centraX/olympusx-contracts** repository was done by Pashov Audit Group, during which **pontifex**, **0k**, **0x37**, **X0sauce**, **BenRai** engaged to review **OLY**. A total of **66** issues were uncovered.

| Protocol Summary | |
|---|---|
| **Project Name** | OLY |
| **Protocol Type** | Yield management |
| **Timeline** | December 31st 2025 - January 20th 2026 |

**Review commit hash:**
- febcd5429b1727467d685da82caa80770f6c7b97 (De-centraX/olympusx-contracts)

**Fixes review commit hash:**
- ad726a74396713024dc9b068624c1ca325564fce (De-centraX/olympusx-contracts)

### Scope

`DefaultValues.sol` `FarmKeeperRewardsLib.sol` `FarmKeeperV3Lib.sol` `FarmKeeperV4Lib.sol` `FarmKeeperValidationLib.sol` `FarmKeeperViewLib.sol` `Farms.sol` `GracePeriodPenalties.sol` `MarketCapLib.sol` `OlympusXHookLib.sol` `StakingCalculations.sol` `SwapActionsLib.sol` `Common.sol` `EthPrice.sol`

`FarmKeeper.sol` `LaunchManager.sol` `OlympusX.sol` `OlympusXAuction.sol` `OlympusXHook.sol` `StakingVault.sol` `SwapActions.sol` `TaxRouter.sol` `ValidatorVault.sol`

# 7. Findings

## Findings count

| Severity | Amount |
|---|---|
| Critical | 5 |
| High | 9 |
| Medium | 25 |
| Low | 27 |
| **Total findings** | **66** |

## Summary of findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [C-01] | Genesis rewards will be stolen | Critical | Resolved |
| [C-02] | Underflow during `FarmKeeper` V4 liquidity minting | Critical | Resolved |
| [C-03] | V4 farm tokenId can be cached but not minted | Critical | Resolved |
| [C-04] | Intermediary `poolId` not updated for nonnative farms | Critical | Resolved |
| [C-05] | Incorrect handling of `stETH` shares causes accounting error in `ValidatorVa` | Critical | Resolved |
| [H-01] | Fail to unwrap WETH in `swapV3EthForFarmToken` | High | Resolved |
| [H-02] | Unused amount may be calculated incorrectly | High | Resolved |
| [H-03] | Limit order incorrectly filled on tick movement | High | Resolved |
| [H-04] | Inefficient liquidity provision due to decimal-agnostic token balance comparison | High | Resolved |
| [H-05] | First depositor can claim all auction tokens before auction ends | High | Resolved |
| [H-06] | Liquidity addition vulnerable to sandwich attacks due to spot price manipulation | High | Resolved |
| [H-07] | Asymmetric unit scaling in `swapAmount` calculation breaks rebalancing logic | High | Resolved |

| ID | Title | Severity | Status |
|---|---|---|---|
| [H-08] | Exact output swaps incur lower tax than exact input swaps due to tax calculation | High | Resolved |
| [H-09] | Withdraw pays only one token while accounting debits both in filled epochs | High | Resolved |
| [M-01] | Reward loss due to precision underflow in cycle payouts | Medium | Resolved |
| [M-02] | Share rate increase causing staking value loss | Medium | Resolved |
| [M-03] | Reward claim pattern issue with blacklisted tokens | Medium | Resolved |
| [M-04] | Eth pricing fallback misprices USD value | Medium | Resolved |
| [M-05] | Overflow risk in price calculation during V4 pool creation | Medium | Resolved |
| [M-06] | Inflation pool accounting gap allows excess distribution | Medium | Resolved |
| [M-07] | Tokens permanently locked in accounting if auction receives no deposit | Medium | Resolved |
| [M-08] | Accrued fees can be exploited to obtain subsidized liquidity positions | Medium | Resolved |
| [M-09] | New limit order participants unfairly receive fees from exited positions | Medium | Resolved |
| [M-10] | Withdrawal requests revert when combined amounts exceed Lido's `MAX_WITHDRAWAL` | Medium | Resolved |
| [M-11] | Imprecise maturity calculation allows premature stake ending | Medium | Resolved |
| [M-12] | Incorrect validation in `validateFarmSetup` allows non-reward tokens in V4 farms | Medium | Resolved |
| [M-13] | Missing validation allows intermediary pool with wrong token causing fund loss | Medium | Resolved |
| [M-14] | Reward debt heuristic can drop rewards after reward token readdition | Medium | Resolved |
| [M-15] | Excess ETH not refunded when placing limit orders | Medium | Resolved |
| [M-16] | Unequal reward distribution lets short-term stakers access long-term rewards | Medium | Acknowledged |
| [M-17] | Inflation reward calculation is not accurate | Medium | Resolved |

| ID | Title | Severity | Status |
|---|---|---|---|
| [M-18] | Ethereum can be locked if required liquidity is not achieved | Medium | Acknowledged |
| [M-19] | Emergency withdrawal fails to recover ETH in `pendingEth` leaving funds in `V` | Medium | Resolved |
| [M-20] | Removed tokens lead to unclaimable rewards causing permanent loss of user funds | Medium | Resolved |
| [M-21] | Missing slippage protection in `mint` function exposes users to token loss | Medium | Resolved |
| [M-22] | Dead stake violates zero staker invariant and absorbs inflation share | Medium | Resolved |
| [M-23] | LP fees can be permanently locked in nonfilled epochs after all users call | Medium | Resolved |
| [M-24] | Missing collect accrued swap fees in `unlockCallbackPlace` | Medium | Resolved |
| [M-25] | Protocol rebalances incur `OLY` sell tax when swapping via taxed v4 farm pool | Medium | Resolved |
| [L-01] | Stake position flooding risk | Low | Resolved |
| [L-02] | Ineffective staking allowance management in `LaunchManager` due to hardcoding | Low | Resolved |
| [L-03] | Claimable amount underflow causing stuck token | Low | Acknowledged |
| [L-04] | `EthPrice` oracle missing price deviation check | Low | Acknowledged |
| [L-05] | Mint transactions vulnerable to griefing via front running due to no partials | Low | Resolved |
| [L-06] | `ValidatorVault.harvest()` reverts on buy and burn rewards below Lido minimum | Low | Resolved |
| [L-07] | `StakeETH` allows ETH to be staked while minting zero `wstETH` | Low | Resolved |
| [L-08] | Unused token allowance after partial swap execution | Low | Resolved |
| [L-09] | Payout schedule advances only one cycle per call when cycles are overdue | Low | Resolved |
| [L-10] | `CyclePayoutTriggered` logs `ethReward = 0` due to reward token ordering issue | Low | Resolved |

| ID | Title | Severity | Status |
|---|---|---|---|
| [L-11] | No `wstETH/stETH` oracle on ETH mainnet | Low | Resolved |
| [L-12] | Self-referral allows users to claim both minter and referrer bonuses | Low | Resolved |
| [L-13] | Farm removal fee distribution mismatch with documentation | Low | Resolved |
| [L-14] | Incorrect liquidity removed value in `FarmKeeper_Removed` event | Low | Resolved |
| [L-15] | Constructor does not validate price configuration parameters | Low | Resolved |
| [L-16] | Intermediary pool configuration lacks validation | Low | Resolved |
| [L-17] | Twap period can be set to an invalid value | Low | Resolved |
| [L-18] | Reward tokens become permanently locked when removed with no active share | Low | Resolved |
| [L-19] | Rounding dust from reward distribution becomes permanently locked | Low | Resolved |
| [L-20] | Summed reward claims across different tokens produce meaningless values | Low | Resolved |
| [L-21] | Inconsistent slippage handling for increase and decrease of liquidity in v4 l | Low | Resolved |
| [L-22] | Implicit assumption on Chainlink price feed decimals in `EthPrice` | Low | Acknowledged |
| [L-23] | `StakeETH` can be front run with a small amount to disrupt user transactions | Low | Resolved |
| [L-24] | Missed slippage protection when users sell OLY | Low | Acknowledged |
| [L-25] | Mint accounting break with mixed liquidity and vesting configuration | Low | Acknowledged |
| [L-26] | Insufficient observation cardinality validation in `FarmKeeper.setTwapPeriod` | Low | Resolved |
| [L-27] | ValidatorVault assumes `stETH` is pegged 1 to 1 with `ETH` | Low | Acknowledged |

# Critical findings

## [C-01] Genesis rewards will be stolen

### Severity

**Impact:** High

**Likelihood:** High

### Description

In the mint phase, users mint OLY tokens via native ether. These native ethers will be sent to the router for distribution.

The distribution part for `genesis` will be held in TaxRouter. The `genesis` team can withdraw these rewards via the function `withdrawEth`.

In function `distributeEth/distributeEthFromMint`, users can trigger these two functions to distribute rewards. In function `_distributeEthInternal`, if the input value is zero, the function will distribute the current native ether balance in this contract. The problem here is that the current native ether balance belongs to `genesis`. This will cause anyone to be able to redistribute `genesis`' rewards to `stakers, validator vault, etc`.

```
    function _distributeMintEth() internal {
            toGenesis = (msg.value * MINT_ETH_TO_GENESIS_BPS) / BPS;
            toTaxRouter = msg.value - toGenesis;
            TaxRouter(payable(taxRouter)).distributeEthToGenesis{value: toGenesis}();
            // Here we will not distribute to genesis again.
            TaxRouter(payable(taxRouter)).distributeEthFromMint{value: toTaxRouter}();
    }
    function distributeEthToGenesis() external payable {
        userToEth[genesisWallet] += msg.value;
        totalToGenesis += msg.value;
        emit EthDistributed(msg.value, 0, 0, 0, 0, msg.value);
    }
    function distributeEth() public payable nonReentrant {
        _distributeEthInternal(BUY_AND_BURN_BPS);
    }

    function _distributeEthInternal(uint256 buyAndBurnBps) internal {
@>        uint256 total = msg.value > 0 ? msg.value : address(this).balance;
        if (total == 0) revert TaxRouter__ZeroAmount();

        // Calculate distributions
        uint256 toStakers = (total * STAKERS_BPS) / BPS;
        uint256 toBuyAndBurn = (total * buyAndBurnBps) / BPS;
        uint256 toValidators = (total * VALIDATORS_BPS) / BPS;
        uint256 toFarms = (total * FARMS_BPS) / BPS;
        uint256 toGenesis = total - (toStakers + toBuyAndBurn + toValidators + toFarms);
```

```
        (bool success,) = stakingVault.call{value: toStakers}("");
        if (!success) revert TaxRouter__TransferFailed();

        if (toBuyAndBurn > 0) {
            (success,) = buyAndBurn.call{value: toBuyAndBurn}("");
            if (!success) revert TaxRouter__TransferFailed();
        }

        if (toValidators > 0) {
            (success,) = validatorVault.call{value: toValidators}("");
            if (!success) revert TaxRouter__TransferFailed();
        }

        if (toFarms > 0) {
            (success,) = farmKeeper.call{value: toFarms}("");
            if (!success) revert TaxRouter__TransferFailed();
        }

        if (toGenesis > 0) {
            userToEth[genesisWallet] += toGenesis;
        }

        // Update statistics
        totalDistributed += total;
        totalToStakers += toStakers;
        totalToBuyAndBurn += toBuyAndBurn;
        totalToValidators += toValidators;
        totalToFarms += toFarms;
        totalToGenesis += toGenesis;

        emit EthDistributed(total, toStakers, toBuyAndBurn, toValidators, toFarms,
toGenesis);
    }
```

### Recommendations

The current native ether in the TaxRouter contract should not be redistributed.

## [C-02] Underflow during `FarmKeeper` V4 liquidity minting

### Severity

**Impact:** High

**Likelihood:** High

### Description

When adding or minting liquidity into V4 liquidity pools tied to farms through `FarmKeeperV4Lib::addLiquidity()`, it is observed that the protocol checks the amount utilized by the `FarmKeeper` contract to increase the LP position before and after the `PositionManager::modifyLiquidities()` call.

However, when reviewing how `unlockData` is prepared via `FarmKeeperV4Lib::_prepareMintPositionUnlockData()` and `FarmKeeperV4Lib::_prepareAddLiquidityUnlockData()`, we observe that the protocol performs a sweep for any remaining native ETH if oversettlement occurs.

Under the hood, this sweep operation calls `PositionManager::_sweep()`, which transfers any remaining token balances held by the `PositionManager` back to the `FarmKeeper` contract.

```
    function _sweep(Currency currency, address to) internal {
@>        uint256 balance = currency.balanceOfSelf();
        if (balance > 0) currency.transfer(to, balance);
    }
```

```
    function balanceOfSelf(Currency currency) internal view returns (uint256) {
        if (currency.isAddressZero()) {
@>            return address(this).balance;
        } else {
            return IERC20Minimal(Currency.unwrap(currency)).balanceOf(address(this));
        }
    }
```

Next, we observe that the calculation used to derive `amount0`, and subsequently `farm.currency0Balance`, is performed inside an unchecked block. This means that if an underflow occurs, the value will wrap around instead of reverting. Specifically for `amount0`, if a malicious user donates `x + 1 wei` worth of ETH to the `PositionManager`, where `x` represents the amount of ETH used to increase the LP position for `FarmKeeper` in the V4 pool, an underflow can be triggered. This is because the `SWEEP` action would transfer the donated ETH to the `FarmKeeper` contract. As a result, the accounting for `amount0` underflows and becomes a very large value, and when this value is subtracted from `farm.currency0Balance`, it causes `farm.currency0Balance` to also underflow and represent an extremely large value.

```
    function _prepareAddLiquidityUnlockData(Farm storage farm, uint128 liquidity)
        internal
        view
        returns (bytes memory unlockData)
    {
        bytes[] memory params = new bytes[](3);
        params[0] = abi.encode(farm.lp.tokenId, liquidity, farm.currency0Balance,
farm.currency1Balance, "");
        params[1] = abi.encode(farm.poolKey.currency0, farm.poolKey.currency1);
        params[2] = abi.encode(CurrencyLibrary.ADDRESS_ZERO, address(this));

        unlockData = abi.encode(
@>            abi.encodePacked(uint8(Actions.INCREASE_LIQUIDITY),
uint8(Actions.SETTLE_PAIR), uint8(Actions.SWEEP)),
            params
        );
    }
```

```
    function addLiquidity(address v4PositionManager, address v4PoolManager, Farm storage
farm, uint48 deadline)
        public
        returns (uint128 liquidity, uint256 amount0, uint256 amount1)
    {
        // ...

        unchecked {
@>            amount0 = token0BalanceBefore - farm.poolKey.currency0.balanceOfSelf();
            amount1 = token1BalanceBefore - farm.poolKey.currency1.balanceOfSelf();

@>            farm.currency0Balance -= amount0;
            farm.currency1Balance -= amount1;
            farm.lp.liquidity = farm.lp.liquidity + liquidity;
        }
    }
```

This issue impacts all subsequent liquidity-related operations, including adding liquidity and depositing into the farm, which may involve swapping ETH to `currency1`. All future liquidity actions will be blocked, since the amounts specified will not actually exist within the `FarmKeeper` contract.

### Recommendations

Consider a different approach in calculating the amount of native ETH used to supply to V4 pools to increase LP positions.

## [C-03] V4 farm tokenId can be cached but not minted

### Severity

**Impact:** High

**Likelihood:** High

### Description

V4 pools are enabled through the `FarmKeeper::enableV4Farm()` call. Once the native pool is enabled, ETH is swapped for the equivalent `currency1` through `_depositToV4Farm()` followed by `_swapV4Currency()`. After that, any remaining `currency0Balance` and `currency1Balance` are supplied to the relevant V4 pool as LP through `_addV4Liquidity()`, which calls `FarmKeeperV4Lib::addLiquidity()`.

The issue lies in the `FarmKeeperV4Lib::addLiquidity()` logic. We observe that when the calculated liquidity to be supplied is 0, the `farm.lp.tokenId`, which represents the ERC6909 position token for the farm, is still cached even though the `FarmKeeper` does not actually mint a position token. In this case, `PositionManager::modifyLiquidities()` is not called and the function returns early. This happens because the unlock data is not prepared and the minting process never occurs, yet the tokenId is still stored.

```
    function addLiquidity(address v4PositionManager, address v4PoolManager, Farm storage
farm, uint48 deadline)
        public
        returns (uint128 liquidity, uint256 amount0, uint256 amount1)
    {
        // ...
        if (farm.lp.tokenId == 0) {
@>            farm.lp.tokenId = IPositionManager(v4PositionManager).nextTokenId();
            (unlockData, liquidity) = _prepareIncreaseUnlockDate(farm, currentTick, true);
        } else {
            (unlockData, liquidity) = _prepareIncreaseUnlockDate(farm, currentTick, false);
        }

@>        if (liquidity == 0) {
@>           return (0, 0, 0);
@>        }

        uint256 token0BalanceBefore = farm.poolKey.currency0.balanceOfSelf();
        uint256 token1BalanceBefore = farm.poolKey.currency1.balanceOfSelf();
@>        IPositionManager(v4PositionManager).modifyLiquidities{value: valueToPass}
(unlockData, deadline);
        // ...
```

```
    function _prepareIncreaseUnlockDate(Farm storage farm, int24 currentTick, bool isMint)
        internal
        view
        returns (bytes memory unlockData, uint128 liquidity)
    {
        int24 tickLower = TickMath.minUsableTick(farm.poolKey.tickSpacing);
        int24 tickUpper = TickMath.maxUsableTick(farm.poolKey.tickSpacing);
        liquidity = LiquidityAmounts.getLiquidityForAmounts(
            TickMath.getSqrtPriceAtTick(currentTick),
            TickMath.getSqrtPriceAtTick(tickLower),
            TickMath.getSqrtPriceAtTick(tickUpper),
@>            farm.currency0Balance,
@>            farm.currency1Balance
        );

@>        liquidity = uint128(Math.mulDiv(liquidity, 9_950, BPS)); // Avoid slippage issues
caused by LiquidityAmounts.getLiquidityForAmounts

@>        if (liquidity == 0) {
@>            return (bytes(""), 0);
        }

        if (isMint) {
            return (_prepareMintPositionUnlockData(farm, tickLower, tickUpper, liquidity),
liquidity);
        } else {
            return (_prepareAddLiquidityUnlockData(farm, liquidity), liquidity);
        }
        // ...
```

As a result, a malicious user can wait for a V4 pool to be enabled by the owner, then wait until `farm.currency0Balance` and `farm.currency1Balance` result in a very small liquidity value that rounds down to 0. Since no liquidity is supplied by the `FarmKeeper` when liquidity

equals 0, no actions are prepared for minting the position token through `_prepareMintPositionUnlockData()`. The user can then add liquidity in a way that causes the position ID to be cached without the `FarmKeeper` actually owning that token.

This leads to all subsequent attempts to add liquidity being blocked because the `FarmKeeper` does not own the cached tokenId, and the `PositionManager::onlyIfApproved()` modifier will cause the revert.

The only recovery path would be for the admin to remove the farm, which would cause the farm to lose ETH allocations and potential yield, since ETH yield distribution to the `FarmKeeper` and all farms only occurs in the `receive()` call.

```
    function _increase(
        uint256 tokenId,
        uint256 liquidity,
        uint128 amount0Max,
        uint128 amount1Max,
        bytes calldata hookData
@>    ) internal onlyIfApproved(msgSender(), tokenId) {
        (PoolKey memory poolKey, PositionInfo info) = getPoolAndPositionInfo(tokenId);

        // Note: The tokenId is used as the salt for this position, so every minted position
has unique storage in the pool manager.
        (BalanceDelta liquidityDelta, BalanceDelta feesAccrued) =
            _modifyLiquidity(info, poolKey, liquidity.toInt256(), bytes32(tokenId),
hookData);
        // Slippage checks should be done on the principal liquidityDelta which is the
liquidityDelta - feesAccrued
        (liquidityDelta - feesAccrued).validateMaxIn(amount0Max, amount1Max);
    }
```

### Recommendations

The recommendation is to avoid caching the next token ID in `farm.lp.tokenId` when no minting occurs due to liquidity being zero.

## [C-04] Intermediary `poolId` not updated for nonnative farms

### Severity

**Impact:** High

**Likelihood:** High

## Description

For farms with non-native pools, the admin adds an intermediary pool to swap ETH allocated to the farm into one of the tokens associated with that farm. This intermediary pool can be configured as either a V3 pool or a V4 pool, and `FarmKeeper::setIntermediaryConfiguration()` allows the admin to update the intermediary pool configuration.

```
    function setIntermediaryConfiguration(
        bytes32 id,
        uint16 slippage,
        uint32 twapPeriod,
        PoolKey memory poolKey,
        bool isPoolV4
    ) external onlyOwner validFarmId(id) {
        FarmKeeperValidationLib.validateSwapSlippage(slippage);
        FarmKeeperValidationLib.validateTwapPeriod(twapPeriod);

        PoolData storage intermediaryPool = intermediaryPools[id];
        if (intermediaryPool.swapSlippage != slippage) {
            intermediaryPool.swapSlippage = slippage;
        }
        if (intermediaryPool.twapPeriod != twapPeriod) {
            intermediaryPool.twapPeriod = twapPeriod;
        }
        if (intermediaryPool.isPoolV4 != isPoolV4) {
            intermediaryPool.isPoolV4 = isPoolV4;
        }
        intermediaryPool.poolKey = poolKey;
        emit FarmKeeper_IntermediaryPoolUpdated(id, slippage, twapPeriod, poolKey, isPoolV4);
```

The issue is that while all relevant fields of the `PoolData::intermediaryPool` struct are updated, the `poolId` is not updated. As a result, slippage will be wrongly calculated in `SwapActionsLib.swapExactInV3()`, which can cause a loss for the farm.

```
    function _swapV3EthForFarmToken(Farm storage farm, PoolData memory intermediaryPool)
private {
        uint256 ethAmount = farm.nativeBalance;
        if (ethAmount == 0) return;

        // Get WETH and intermediary token addresses
        (address wethToken, address intermediaryToken) =
FarmKeeperV3Lib.getSwapTokens(intermediaryPool, isWethToken);

        // Wrap ETH to WETH
        IWETH9(wethToken).deposit{value: ethAmount}();

        SwapExactInResult memory result = _swapV3Token(
            SwapActions.SwapExactInParams({
                amountIn: ethAmount, slippage: intermediaryPool.swapSlippage, twapPeriod:
intermediaryPool.twapPeriod
            }),
@>            intermediaryPool.poolId,
            intermediaryPool.poolKey.fee,
            wethToken,
```

```
            intermediaryToken
        );

        // Update balances using library
        FarmKeeperV3Lib.processEthSwapResult(farm, intermediaryPool, result, isWethToken);

        emit FarmKeeper_NativeSwapped(farm.id, ethAmount, result.amountOut);
    }
```

```
    function swapExactInV3(
        ISwapRouter02 swapRouter,
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint24 fee,
@>        uint160 sqrtPriceX96,
        uint16 slippageBPS
    ) public returns (uint256 amountOut) {
        IV3SwapRouter.ExactInputSingleParams memory params =
IV3SwapRouter.ExactInputSingleParams({
            tokenIn: tokenIn,
            tokenOut: tokenOut,
            fee: fee,
            recipient: address(this),
            amountIn: amountIn,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: (tokenIn < tokenOut ? MIN_SQRT_RATIO : MAX_SQRT_RATIO)
        });

@>        amountOut = swapRouter.exactInputSingle(params);

        require(
@>            amountOut >= _calculateAmountOutMinimum(amountIn, sqrtPriceX96, tokenIn,
tokenOut, slippageBPS),
            SwapActionsLib__TooLittleReceived()
        );
    }
```

Consider the following scenario for a set of token0 <> token1 pools.

- A farm uses a V4 `WBTC <> USDC` pool with a V3 intermediary pool `WETH <> WBTC`.
- The intermediary pool is changed from `WETH <> WBTC` to `WETH <> USDC`.
- The `poolId` is not updated and still refers to the `WETH <> WBTC` pool.

As a result, the swap logic continues to treat the swap as `WETH <> WBTC`, causing the slippage calculated to use the `sqrtPriceX96` representing 1 WBTC in terms of WETH instead of 1 WBTC in terms of USDC.

## Recommendations

The recommendation is to also update the corresponding `poolId` whenever a new `intermediaryPool` configuration is set.

# [C-05] Incorrect handling of `stETH` shares causes accounting error in `ValidatorVa`

## Severity

**Impact:** High

**Likelihood:** High

## Description

The `stakeETH()` function in `ValidatorVault` incorrectly treats the return value of `stETH.submit()` as the amount of stETH tokens received, when it actually returns the number of shares minted. This fundamental misunderstanding of Lido's rebasing token mechanism leads to systematic under-wrapping of stETH to wstETH, causing multiple critical issues with fund accounting and reward distribution.

**Root Cause:**

According to Lido's documentation, `stETH.submit()` returns the number of shares minted, not the stETH token amount. The relationship between shares and stETH tokens is:

```
stETH_balance = shares * totalPooledEther / totalShares
```

Currently, the share-to-stETH ratio is approximately 1 share ≈ 1.22 stETH (and growing over time as staking rewards accrue). The code incorrectly uses the share count as the amount of stETH to wrap:

```
uint256 stETHReceived = stETH.submit{value: amount}(address(0)); // Returns shares, not tokens
stETH.approve(wstEth, stETHReceived); // Approves too little
uint256 wstEthReceived = wstETHContract.wrap(stETHReceived); // Wraps based on shares, not actual stETH balance
```

Concrete Example:

When a user stakes 1 ETH:

1. stETH.submit{value: 1 ether}() returns approximately 0.82 shares.
2. The actual stETH token balance received is approximately 1.0 stETH (calculated as 0.82 * 1.22 ≈ 1.0).
3. The code approves and wraps only 0.862 (treating shares as tokens).
4. Approximately 0.18 stETH (approximately 18%) remains unwrapped in the contract.

Critical Consequences:

1. Trapped stETH: Each staking operation leaves unwrapped stETH in the contract. This accumulates over time and is never accounted for in the protocol's tracking variables (totalWstETHStaked, totalWstETHPrincipal, totalETHStaked). The trapped stETH continues to rebase and earn rewards, but these rewards are completely inaccessible.
2. Broken Reward Distribution: The harvest() function calculates rewards as the difference between getWstEthValue() and totalWstETHPrincipal. Since getWstEthValue() only considers totalWstETHStaked (which does not include the trapped stETH which is approximately 18% of all staked ETH), the calculation systematically underestimates the actual value held by the vault. This means:
    - Legitimate staking rewards from wstETH are nearly never distributed since 18% of the value is not accounted for
    - The trapped stETH's value appreciation is never harvested
    - Eventually, currentEthValue > totalWstETHPrincipal may hold true but the distributed rewards will be way too little
3. Inaccurate Principal Tracking: The variable totalWstETHPrincipal is incremented by the full ETH amount staked, but totalWstETHStaked is incremented by an amount that does not reflect the total value received from Lido. This creates a permanent accounting mismatch where principal is greater than actual tracked assets.
4. Emergency Withdrawal Issue: The emergencyWithdrawWstEth() function only withdraws wstETH tracked in totalWstETHStaked. Any accumulated unwrapped stETH remains permanently trapped in the contract with no recovery mechanism.
5. Growing Impact: As the stETH share-to-token ratio increases over time (due to staking rewards), the percentage of trapped funds grows, making the issue progressively worse with each stake operation.

## Recommendation

Modify the `stakeETH()` function to use the actual stETH token balance instead of the shares returned by `submit()` by determining the stETH balance held by the contract approving it and using it in the call to wrap:

```
uint256 stETHReceived = stETH.balanceOf(address(this));
stETH.approve(wstEth, stETHReceived);

// Wrap stETH to wstEth
uint256 wstEthReceived = wstETHContract.wrap(stETHReceived);
```

This ensures that all stETH received from Lido is properly wrapped into wstETH, preventing trapped funds and maintaining accurate accounting throughout the vault's operations.

# High findings

## [H-01] Fail to unwrap WETH in `swapV3EthForFarmToken`

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

In FarmKeeper, users can deposit assets to Uniswap pools via `depositToFarm`. If the target pool is not one native pool, the native ether will be swapped to an intermediary token via function `_swapNativeForFarm` at first.

If the intermediaryPool is one Uniswap v3 pool, `nativeBalance` will be wrapped to WETH at first. It is possible that there will be some remaining WETH, recorded by `swapResult.unusedAmount`. These unused WETH will be recorded back to `nativeBalance`. In the next farm, these native ether can be used again.

The problem here is that the `unused` WETH fails to be unwrapped to native Ether. This will cause that in the next farm, the actual native ether might be less than `nativeBalance`. The farm transaction will be reverted.

```
    function _swapV3EthForFarmToken(Farm storage farm, PoolData memory intermediaryPool)
private {
        // fetch this farm's native Ether amount.
        uint256 ethAmount = farm.nativeBalance;
        if (ethAmount == 0) return;

        // Get WETH and intermediary token addresses
        // In this intermediary pool, WETH/TokenX.
        (address wethToken, address intermediaryToken) =
FarmKeeperV3Lib.getSwapTokens(intermediaryPool, isWethToken);
        // Wrap ETH to WETH
        // Convert all native Balance to WETH.
        IWETH9(wethToken).deposit{value: ethAmount}();

        SwapExactInResult memory result = _swapV3Token(
            SwapActions.SwapExactInParams({
                amountIn: ethAmount, slippage: intermediaryPool.swapSlippage, twapPeriod:
intermediaryPool.twapPeriod
            }),
            intermediaryPool.poolId,
            intermediaryPool.poolKey.fee,
            wethToken,
            intermediaryToken
        );
        FarmKeeperV3Lib.processEthSwapResult(farm, intermediaryPool, result, isWethToken);
```

```
        emit FarmKeeper_NativeSwapped(farm.id, ethAmount, result.amountOut);
    }
```

### Recommendations

Unwrap the unused WETH to native Ether in `_swapV3EthForFarmToken`.

## [H-02] Unused amount may be calculated incorrectly

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

In the SwapActions contract, there is one swap retry mechanism. If this swap slippage is larger than expected, the swap action will retry until the retry count reaches `maxRetries`.

The problem here is that if the swap fails for all retries, the actual used amount should be 0. However, the `unusedAmount` will be calculated by `result.unusedAmount = params.amountIn - amountToSwap`. This will cause the `unusedAmount` to be less than expected.

```
    function _swapV4Currency(
        SwapExactInParams memory params,
        PoolKey memory poolKey,
        Currency currencyIn,
        Currency currencyOut,
        uint48 deadline
    ) internal returns (SwapExactInResult memory result) {
        // true --> isV4 Pool.
        _approveTokenIfNeeded(Currency.unwrap(currencyIn), address(v4Router),
params.amountIn, true, deadline);

        result.twapSqrtPriceX96 = V4OracleLib.getTwapSqrtPriceX96(poolKey,
params.twapPeriod);
        uint128 amountToSwap = uint128(params.amountIn);

        uint8 retryCount = 0;

        SWAP_RECEIVABLE_SLOT.asBoolean().tstore(true);
        while (amountToSwap > 0 && retryCount < maxRetries) {
            try SwapActionsLib.swapExactInV4(
                IUniversalRouter(v4Router),
                poolKey,
                currencyIn,
                currencyOut,
                amountToSwap,
                result.twapSqrtPriceX96,
```

```
                params.slippage
            ) returns (
                uint256 amountOut
            ) {
                result.amountOut = amountOut;
                break;
            } catch (bytes memory reason) {
                amountToSwap = uint128(_getNewSwapAmountForRetry(reason, amountToSwap,
swapAmountForRetryBPS));
            }
            unchecked {
                ++retryCount;
            }
        }
        SWAP_RECEIVABLE_SLOT.asBoolean().tstore(false);

@>        result.unusedAmount = params.amountIn - amountToSwap;
    }
```

### Recommendations

The accrued fees should be distributed to the last limit order.

## [H-03] Limit order incorrectly filled on tick movement

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

The `OlympusXHook` contract relies on the `_getCrossedTicks()` function to identify which ticks were crossed during a swap by comparing the current pool tick with the previously recorded tick. Based on this range, `_afterSwap()` processes any limit orders whose tick ranges are considered crossed and marks them as filled.

However, this approach can incorrectly mark limit orders as fully filled in certain price movement patterns. Specifically, if a swap moves the tick into a limit order's range by crossing the lower boundary but does not cross the upper boundary, and a subsequent swap then moves the tick back in the opposite direction, the limit order may be incorrectly treated as filled.

Consider the following scenario:

- current tick = -200
- tick spacing = 200
- limit order tick range = [0, 200]
- A swap moves the tick from -200 to 100. This crosses the lower boundary at tick 0, and `_afterSwap()` updates `tickLowerLast` to 0.

```
    function _afterSwap(
        address sender,
        PoolKey calldata key,
        SwapParams calldata params,
        BalanceDelta delta,
        bytes calldata
    ) internal override returns (bytes4, int128) {
        // Handle limit order fills
@>        (int24 tickLower, int24 lower, int24 upper) = _getCrossedTicks(key.toId(),
key.tickSpacing);
        // ...

@>        _setTickLowerLast(key.toId(), tickLower);
        return (IHooks.afterSwap.selector, taxDelta);
    }
```

1. A subsequent swap moves the tick from 100 to -100. The tick decreases and crosses the lower boundary of the limit order range again.

```
    function _afterSwap(
        address sender,
        PoolKey calldata key,
        SwapParams calldata params,
        BalanceDelta delta,
        bytes calldata
    ) internal override returns (bytes4, int128) {
        // Handle limit order fills
@>        (int24 tickLower, int24 lower, int24 upper) = _getCrossedTicks(key.toId(),
key.tickSpacing);
        // ...

@>        _setTickLowerLast(key.toId(), tickLower);
        return (IHooks.afterSwap.selector, taxDelta);
    }
```

1. `OlympusXHook::_getCrossedTicks()` computes:
2. `tickLower` = -100
3. `tickLowerLast` = 0
4. returned crossed range = (-100, 0, 0)

```
    function _getCrossedTicks(PoolId poolId, int24 tickSpacing)
    private
    view
```

```
    returns (int24 tickLower, int24 lower, int24 upper)
{
    (, int24 tick,,) = poolManager.getSlot0(poolId);
@>    tickLower = _getTickLower(tick, tickSpacing);
    int24 tickLowerLast = tickLowerLasts[poolId];

@>    if (tickLower < tickLowerLast) {
@>        lower = tickLower + tickSpacing;
@>        upper = tickLowerLast;
    } else {
        lower = tickLowerLast;
        upper = tickLower - tickSpacing;
    }
}
```

1. The loop in `_afterSwap()` then treats the limit order at tick range [0, 200] as filled, calling `_fillEpoch()` and removing the order from the pool.

```
        address sender,
        PoolKey calldata key,
        SwapParams calldata params,
        BalanceDelta delta,
        bytes calldata
    ) internal override returns (bytes4, int128) {
        // Handle limit order fills
@>        (int24 tickLower, int24 lower, int24 upper) = _getCrossedTicks(key.toId(),
key.tickSpacing);
        if (lower <= upper) {
            // note that a zeroForOne swap means that the pool is actually gaining token0, so
limit
            // order fills are the opposite of swap fills, hence the inversion below
            bool zeroForOne = !params.zeroForOne;
            for (; lower <= upper; lower += key.tickSpacing) {
@>                _fillEpoch(key, lower, zeroForOne);
            }
        }

        // ...
    }
    // ...
    function _fillEpoch(PoolKey calldata key, int24 lower, bool zeroForOne) internal {
        Epoch epoch = getEpoch(key, lower, zeroForOne);
        if (!epoch.equals(EPOCH_DEFAULT)) {
            EpochInfo storage epochInfo = epochInfos[epoch];

            // Skip if there's no liquidity to fill
            if (epochInfo.liquidityTotal == 0) {
                return;
            }

@>            epochInfo.filled = true;

            (uint256 amount0, uint256 amount1) =
                _unlockCallbackFill(key, lower, -int256(uint256(epochInfo.liquidityTotal)));

            unchecked {
                epochInfo.token0Total += amount0;
```

```
                epochInfo.token1Total += amount1;
            }

            _setEpoch(key, lower, zeroForOne, EPOCH_DEFAULT);

            emit Fill(epoch, lower, zeroForOne, epochInfo.liquidityTotal, amount0, amount1);
        }
    }
```

This behavior is incorrect because the limit order was only partially filled: the price never crossed the upper boundary of the limit order range at tick 200, which is required for a full fill.

### Recommendations

The recommendation is to introduce explicit handling for partial fills. This would allow users to claim any filled portion of their order while keeping the remaining unfilled portion active until the price fully crosses the limit order's tick range.

## [H-04] Inefficient liquidity provision due to decimal-agnostic token balance comparison

### Severity

**Impact:** Medium

**Likelihood:** High

### Description

The `depositToFarm()` function in `FarmKeeper` determines which token to swap by comparing raw token balances without accounting for token decimals. This breaks the protocol's efficiency guarantee that all available tokens should be optimally deployed as liquidity.

When `depositToFarm()` is called, the function compares `farm.currency0Balance > farm.currency1Balance` to decide which token to swap. This comparison uses raw wei amounts, which are not comparable across tokens with different decimal places. For example, in a USDC (6 decimals) / DAI (18 decimals) pool:

- If the intermediate pool swap provides 100 USDC (1e6) and fees (to be reinvested) accumulate 1 DAI (1e18), the comparison `100 * 1e6 > 1e18` evaluates to false.
- The function incorrectly decides to swap DAI instead of USDC.
- Since 1 DAI is worth significantly less than 100 USDC at parity, the system swaps DAI to USD, increasing the balance of the token that is already more available.
- This leaves up to 10^12 units of USDC dormant and uninvested in the liquidity pool for each DAI available.

The issue manifests when:

1. A farm has tokens with mismatched decimals (e.g., 6 vs 18 in the USDC/DAI pool).
2. Fees accumulate in the higher-decimal token (all pools accumulate fees in both tokens, and part of it is reinvested).
3. An intermediate pool swap provides the lower-decimal token.
4. `depositToFarm()` is called to add liquidity.

### Recommendation

Use the pool's current price to determine which token is in excess. The recommended approach:

1. Get the current pool price from slot0. (e.g. 3000 USDC/ETH).
2. Calculate the value of currency0Balance (ETH) in currency1 (e.g. 1 ETH * 3000 USDC/ETH = 3000 USDC).
3. Compare the result to the currency1Balance. If it is smaller, currency0 is in excess; if it is larger, currency1 is in excess. (e.g. USDC balance of 5000 USDC => 3000 < 5000 => USDC is in excess).
4. Determine the amount of tokens to swap by getting the difference between both values and determining how many tokens to swap (e.g. 5000 - 3000 = 2000 => swap 1000 USDC to ETH).
5. Swap the excess token to achieve optimal liquidity provision.

This also ensures that both existing balances are taken into account and not only the one that is swapped.

## [H-05] First depositor can claim all auction tokens before auction ends

### Severity

**Impact:** Medium

**Likelihood:** High

### Description

The `claim()` function in `OlympusXAuction` allows users to claim their proportional share of OLY tokens without verifying whether the auction for the current cycle has concluded. This creates a critical race condition where the first depositor during an active auction can immediately claim all distributed tokens before other users have a chance to participate.

When a user deposits ETH via `deposit()` during an active auction, their contribution is recorded in `userDeposits[msg.sender][currentCycle]` and added to `totalEthDeposited[currentCycle]`. The `claim()` function then calculates the claimable amount using the formula:

```
(userDeposit * olympusXAmount) / totalEth;
```

However, there is no check in the `claim()` function preventing claims during an ongoing auction. This allows the following attack scenario:

1. Owner calls `startDistribution()` to begin a new cycle, allocating 1,000,000 OLY tokens for distribution over an 8-day auction period.
2. Attacker immediately deposits 1 wei via `deposit()`, becoming the first participant.
3. At this moment: `totalEthDeposited[currentCycle]` = 1 wei.
4. Attacker immediately calls `claim(currentCycle)`, receiving (1 wei * 1,000,000 OLY) / 1 wei = 1,000,000 OLY tokens.
5. The attacker's deposit is set to zero: `userDeposits[attacker][currentCycle]` = 0.
6. All subsequent depositors who participate during the remaining auction period receive nothing, as `olympusXDistributedPerCycle[currentCycle]` remains 1,000,000 OLY but the tokens are already transferred out.

The impact extends beyond a single cycle: if the attacker repeats this pattern (deposit, claim immediately, deposit again), they can continuously drain tokens from previous cycles that have not been claimed yet. Additionally, since totalReservedTokens is decremented on each claim, the accounting becomes corrupted, potentially affecting future distributions.

The function `isAuctionActive()` exists to check auction status, but `claim()` never invokes it to enforce auction completion before allowing claims.

### Recommendation

Add a validation check in the `claim()` function to ensure the auction has ended before allowing claims for the current cycle. This ensures all participants have the full auction duration to deposit before any claims can be processed, maintaining fair proportional distribution.

## [H-06] Liquidity addition vulnerable to sandwich attacks due to spot price manipulation

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

The `addLiquidity()` function in `FarmKeeperV4Lib` calculates the liquidity amounts to add using the current spot price obtained from `getSlot0()`. This creates an opportunity for sandwich attacks where an attacker can manipulate the pool price immediately before liquidity addition to extract value.

The attack flow works as follows:

1. An attacker monitors the amount of liquidity available to be added to a pool.
2. If enough liquidity is available to make the attack profitable, the attacker swaps in the pool to move the current tick to an extreme price.
3. He calls the public `addLiquidity()` function, which reads the manipulated tick via `getSlot0()` and calculates liquidity based on this distorted price.
4. The liquidity is added at unfavorable ratios for the protocol.
5. The attacker swaps back to the original price, profiting from the imbalanced liquidity addition.

This vulnerability is particularly severe because `depositToFarm()` is an external function callable by anyone, meaning an attacker can trigger liquidity additions at will without needing to wait for legitimate users.

Notably, the protocol already implements TWAP-based pricing in the `decreaseLiquidity()` function, which correctly uses `V4OracleLib.getTwapSqrtPriceX96()` for slippage protection.

### Recommendation

Replace the spot price from `getSlot0()` with the TWAP price calculation, consistent with how `decreaseLiquidity()` operates.

This ensures that liquidity additions are based on a time-weighted average price that cannot be manipulated within a single transaction, providing the same level of protection that is already present in the liquidity removal logic.

## [H-07] Asymmetric unit scaling in `swapAmount` calculation breaks rebalancing logic

### Severity

**Impact:** High

**Likelihood:** Medium

## Description

In `FarmKeeper.depositToFarm()`, the swap amount (`swapAmount`) is calculated asymmetrically depending on swap direction. In the `token1 → token0` branch, the balance is multiplied by `1e6`, while in the `token0 → token1` branch, no such scaling is applied. As a result, when `swapToken0 == true`, the computed `swapAmount` is typically **~1e6 times smaller** than intended, effectively disabling rebalancing in that direction.

This causes systematic under-swapping, leaving one side of the pair underfunded and preventing proper liquidity balancing.

**Asymmetric `swapAmount` formula**

```
bool swapToken0 = farm.currency0Balance > farm.currency1Balance;

uint256 swapAmount =
    (swapToken0 ? farm.currency0Balance : farm.currency1Balance * 100_0000)
        / (200_0000 - farm.poolKey.fee);
```

The `100_0000` factor is applied only in one branch, even though both `currency0Balance` and `currency1Balance` represent raw ERC20 amounts.

**Balances are tracked as raw token amounts**

```
if (tokenIn == farm.poolKey.currency0) {
    farm.currency0Balance -= (amountIn - result.unusedAmount);
    farm.currency1Balance = farm.currency1Balance + result.amountOut;
} else {
    farm.currency1Balance -= (amountIn - result.unusedAmount);
    farm.currency0Balance = farm.currency0Balance + result.amountOut;
}
```

**Raw balances are used directly for adding liquidity**

```
prepareAddLiquidityArgs(
    token0,
    token1,
    farm.currency0Balance > type(uint128).max ? type(uint128).max :
uint128(farm.currency0Balance),
    farm.currency1Balance > type(uint128).max ? type(uint128).max :
uint128(farm.currency1Balance),
    twapSqrtPriceX96,
    farm.liquiditySlippage
);
```

Because the scaling factor is applied inconsistently, rebalancing in one direction is almost always negligible, leading to idle balances and reduced liquidity provider utilization.

## Recommendations

Make the `swapAmount` calculation symmetric by using consistent units in both branches. After fixing unit consistency, reassess the intended swap sizing logic separately.

# [H-08] Exact output swaps incur lower tax than exact input swaps due to tax calculation

## Severity

**Impact:** High

**Likelihood:** Medium

## Description

The protocol implements different tax calculation formulas for exact input versus exact output swaps when users sell OLY tokens for ETH, resulting in users paying substantially less tax when using exact output swaps. This creates an exploitable arbitrage opportunity and undermines the protocol's dynamic tax mechanism designed to regulate market activity based on market capitalization.

The tax discrepancy occurs because of different calculation methods in `_beforeSwap()` and `_afterSwap()`:

**Exact Output Tax (in `_beforeSwap()`):**
When `params.amountSpecified > 0`, the user specifies the desired ETH output amount. The tax is calculated as:

```
taxAmount = (netRequested * taxBps) / BPS
```

**Exact Input Tax (in `_afterSwap()`):**
When `params.amountSpecified < 0`, the user specifies the OLY input amount. The tax is calculated after the swap on the actual ETH received:

```
taxAmount = (outputAmount * taxBps) / BPS
```

**Mathematical Impact:**

For a 38% tax rate (TIER1_TAX_BPS), if a user wants to receive 100 ETH net:

- **Exact Output:** `taxAmount = 100 * 38% = 38 ETH`, total output from pool = 138 ETH
- **Exact Input:** We receive 138 ETH total output, `taxAmount = 138 * 38% = 52.44 ETH`, net received = 85.56 ETH

This means to receive the same 100 ETH net, an exact input user would need approximately 161.29 ETH total output (resulting in 61.29 ETH tax), while an exact output user only needs 138 ETH total output (resulting in 38 ETH tax).

The exact output user effectively pays: `38 / 138 ≈ 27.5%` effective tax rate instead of the intended 38%, representing a **27.6% reduction** in tax burden.

**Attack Scenario:**

1. Sophisticated traders will always use exact output swaps to minimize their tax liability.
2. MEV bots and arbitrageurs can exploit this by structuring all their OLY to ETH swaps as exact output.
3. The protocol collects significantly less tax revenue than intended, reducing funds available for buybacks, burns, and ecosystem development.
4. The dynamic tax system (with tiers from 38% down to 8% based on market capitalization) is undermined, as users can artificially reduce their effective tax rate.

### Recommendation

Standardize the tax calculation to ensure the same effective tax rate regardless of swap type. The correct formula for exact output swaps in `_beforeSwap()` should account for the tax being part of the total amount taken from the pool and be calculated as follows:

```
taxAmount = (netRequested * BPS) / (BPS - taxBps)
```

With this fix, for a 38% tax rate and 100 ETH net requested:

- taxAmount = (100 * 3800) / (10000 - 3800) = 380000 / 6200 ≈ 61.29 ETH.
- Total output from pool = 100 + 61.29 = 161.29 ETH.
- Effective tax rate = 61.29 / 161.29 = 38% ✓.

This ensures exact output swaps pay the same effective tax rate as exact input swaps, maintaining the integrity of the protocol's dynamic tax mechanism.

## [H-09] Withdraw pays only one token while accounting debits both in filled epochs

### Severity

**Impact:** Medium

**Likelihood:** High

### Description

When an epoch is filled, the protocol materializes the result in **two tokens** and stores them in `epochInfo.token0Total` and `epochInfo.token1Total`. During `withdraw()`, the user's proportional share is calculated and deducted from **both totals**. However, in `unlockCallbackWithdraw`, the user is paid **only one token**, depending on the `zeroForOne` direction. The second token (including principal and/or fees accumulated in the "unexpected" currency) may not be paid out and can remain locked on the hook as minted claims in `PoolManager`.

This creates a mismatch between accounting and payout: user balances for both tokens are debited, but only one side is actually transferred.

**Fill records both token amounts into epoch totals**

```
(uint256 amount0, uint256 amount1) =
    _unlockCallbackFill(key, lower, -int256(uint256(epochInfo.liquidityTotal)));

unchecked {
    epochInfo.token0Total += amount0;
    epochInfo.token1Total += amount1;
}
```

**Fill callback can mint both tokens to the hook**

```
if (delta.amount0() > 0) {
    poolManager.mint(address(this), key.currency0.toId(), amount0 =
uint128(delta.amount0()));
}
if (delta.amount1() > 0) {
    poolManager.mint(address(this), key.currency1.toId(), amount1 =
uint128(delta.amount1()));
}
```

**Withdraw debits both totals proportionally**

```
amount0 = FullMath.mulDiv(epochInfo.token0Total, liquidity, liquidityTotal);
amount1 = FullMath.mulDiv(epochInfo.token1Total, liquidity, liquidityTotal);

epochInfo.token0Total -= amount0;
epochInfo.token1Total -= amount1;
epochInfo.liquidityTotal = liquidityTotal - liquidity;
```

**Withdraw callback pays only one token**

```
if (zeroForOne) {
    // pays only token1
    if (token1Amount > 0) {
        poolManager.burn(address(this), currency1.toId(), token1Amount);
        currency1.take(poolManager, to, token1Amount, false);
    }
} else {
    // pays only token0 (+ tax)
    uint256 totalAmount0 = token0Amount + taxAmount;
    if (totalAmount0 > 0) {
        poolManager.burn(address(this), currency0.toId(), totalAmount0);
        if (token0Amount > 0) currency0.take(poolManager, to, token0Amount, false);
        if (taxAmount > 0 && taxRouter != address(0)) currency0.take(poolManager, taxRouter,
taxAmount, false);
    }
}
```

Under realistic market conditions (price oscillations within range and bidirectional swap fees), `_unlockCallbackFill` can produce non-zero amounts for **both tokens**. In such cases, the second token becomes a candidate for underpayment and permanent lock, since it is deducted from epoch totals but never transferred during withdrawal.

## Recommendations

Make payout consistent with accounting by either:

- paying out **both** `token0Amount` and `token1Amount` in `unlockCallbackWithdraw` (burning both claims and preserving existing tax logic on sales); or
- enforcing a strict invariant that filled epochs are single-sided **before** recording values into `token0Total` / `token1Total`, so that one-sided withdrawal is always correct.

# Medium findings

## [M-01] Reward loss due to precision underflow in cycle payouts

### Severity

**Impact:** High

**Likelihood:** Low

### Description

It is observed that reward tokens are paid out every X cycle days depending on the cycle number. Each time a token reward payout is triggered through `StakingVault::_triggerCyclePayout()`, the value of `cumulativeRewardPerShare[token][_cycleNo]` is increased.

However, in situations where `_globalActiveShares` is very large and the value of `tokenReward * PRECISION` is very small, an underflow can occur. When this happens, `cumulativeRewardPerShare[token][_cycleNo]` is not incremented, while `tokenCyclePayoutPool[token][_cycleNo]` is reset to zero. This issue is more likely for reward tokens with low decimal precision and high value, such as `WBTC`.

```
    function _triggerCyclePayout(uint256 _cycleNo, uint256 _globalActiveShares, uint256
_currentContractDay)
        internal
        returns (PayoutTriggered triggered)
    {
        // ...

        if (_globalActiveShares > 0) {
            // Handle all registered reward tokens (including ETH as address(0))
            for (uint256 i = 0; i < tokensLength; i++) {
                address token = rewardTokens[i];
                uint256 tokenReward = tokenCyclePayoutPool[token][_cycleNo];

                if (tokenReward > 0) {
                    // Add to cumulative reward per share
@>                    cumulativeRewardPerShare[token][_cycleNo] += (tokenReward *
PRECISION) / _globalActiveShares;
                    tokenCyclePayoutPool[token][_cycleNo] = 0; // Clear the pool

                    // Track ETH distribution separately for statistics
                    if (token == address(0)) {
                        totalEthDistributed = totalEthDistributed + tokenReward;
                    }
                }
            }
        }
        // ...
```

As a result, reward tokens can become permanently stuck in the `StakingVault` contract, with no available mechanism to recover them. This leads to an irreversible loss of rewards for vault stakers.

### Recommendations

The recommendation is to use a higher precision when incrementing `cumulativeRewardPerShare` during payout triggers, and then scale the value down when users claim their rewards. However, this approach would require adjusting how `cumulativeRewardPerShare` precision is handled across all related logic in the contract. A simpler fix is to introduce rescue token mechanisms so that the rewards can be redistributed.

## [M-02] Share rate increase causing staking value loss

### Severity

**Impact:** High

**Likelihood:** Low

### Description

It is observed that when a user stakes, the `currentShareRate` can increase due to the daily multiplier of 1.0003 applied to the previous day's share rate through `StakingCalculations::updateShareRate()`.

```
    function _updateShareRate() internal {
        if (currentDay > lastShareRateUpdate) {
            uint40 daysPassed = currentDay - lastShareRateUpdate;
@>            currentShareRate = StakingCalculations.updateShareRate(currentShareRate,
daysPassed, CAPPED_MAX_RATE);
            lastShareRateUpdate = currentDay;
            emit ShareRateUpdated(currentShareRate, currentDay);
        }
    }
```

As a result, users who stake `OLY` tokens in exchange for vault shares may receive fewer vault shares for the same amount staked. This design favors earlier stakers over later stakers.

```
    function calculateShares(uint256 amount, uint16 stakeDays, uint256 currentShareRate)
        internal
        pure
        returns (uint256 shares)
    {
@>        shares = (amount * PRECISION) / currentShareRate;
        uint256 bonusBPS = Math.mulDiv(
            MAX_BONUS_BPS, stakeDays - MIN_STAKE_DAYS, MAX_STAKE_DAYS - MIN_STAKE_DAYS,
Math.Rounding.Floor
        );
```

```
        uint256 bonus = (shares * bonusBPS) / BPS;
        shares += bonus;
    }
```

There are also scenarios where users can experience an actual loss in value, measured in vault shares, for the same amount of `OLY` staked.

1. Users who submit staking transactions close to the time when `currentDay` rolls over may experience a sudden increase in `currentShareRate`, which leads to receiving fewer shares. If the vault has been idle with no staking activity for multiple days, this loss can be even more significant.
2. Users who stake very small amounts of `OLY` may receive zero vault shares due to rounding effects (because `currentShareRate` starts at 800e18).

It is also important to note that `StakingVault` shares are intended to be used as a voting token for future governance proposals. Any loss in share value, therefore, also translates into reduced voting power for the staker.

### Recommendations

The recommendation is to include a slippage mechanism that allows users to specify the minimum number of shares they expect to receive for a given amount of staked `OLY` tokens.

## [M-03] Reward claim pattern issue with blacklisted tokens

### Severity

**Impact:** High

**Likelihood:** Low

### Description

It is observed that for token rewards claimed by users during staking and unstaking, the protocol uses a push pattern rather than a pull pattern for ERC20 tokens. For tokens that include a blacklisting mechanism, this creates an issue where a staker who is blacklisted by USDC cannot open new stakes or close an existing stake.

This occurs because both `StakingVault::_startStake()` and `StakingVault::_processEndStake()` call the internal function `StakingVault::_claimAllRewardsInternal()` -> `StakingVault::_claimAllTokensForCycle()`, which attempts to transfer reward tokens to the user. If the staker is blacklisted for USDC and has accumulated USDC rewards, the token transfer will fail and cause the entire staking or unstaking transaction to revert.

```
    function _claimAllTokensForCycle(address _user, uint256 _cycleNo) internal returns
(uint256 totalClaimed) {
        uint256 tokensLength = rewardTokens.length;
```

```
        // Process all tokens for this cycle
        for (uint256 t = 0; t < tokensLength; t++) {
            address token = rewardTokens[t];

            // Claim rewards for this token in this cycle
            uint256 tokenReward = _claimUserTokenCycleReward(_user, token, _cycleNo);

            // Transfer rewards if any
            if (tokenReward > 0) {
                totalClaimed += tokenReward;
                if (token == address(0)) {
                    pendingEthWithdrawals[_user] = pendingEthWithdrawals[_user] +
tokenReward;
                } else {
@>                      IERC20(token).safeTransfer(_user, tokenReward);
                }
                // ...
        }
    }
```

As a result, the user cannot create new staking positions. In addition, if the user is unable to unstake, they may incur penalties on their principal and inflation rewards after exceeding the seven-day maturity grace period.

### Recommendations

The recommendation is to adopt a pull-based approach for claiming ERC20 rewards, similar to the mechanism currently used for claiming ETH rewards.

## [M-04] Eth pricing fallback misprices USD value

### Severity

**Impact:** High

**Likelihood:** Low

### Description

We observe that in multiple instances within the v4 hook `OlympusXHook` and in `LaunchManager`, the protocol relies on an `EthPrice` contract to price ETH in terms of USD.

By default, this contract retrieves the USD price using the Chainlink ETH to USD price oracle, and the resulting value is scaled to 18 decimal places.

However, if the price feed is stale or returns an abnormal value, the protocol falls back to using a TWAP price from Uniswap. This can be sourced from any Uniswap V3 or Uniswap V4 pool with sufficient liquidity, where the returned square root price is squared to derive the price of token0 in terms of token1.

For simplicity, we consider the Uniswap V3 WETH to USDT pool that is configured in the protocol tests, although the same issue also applies to Uniswap V 4 pools.

```
    function getEthUSDPoolV3ByChainId(uint256 chainId) internal pure returns (address) {
        //Ethereum
        if (chainId == 1) {
@>            return address(0x4e68Ccd3E89f51C3074ca5072bbAC773960dFa36);
        }
        //Sepolia
        else if (chainId == 11155111) {
            return address(0x3289680dD4d6C10bb19b899729cda5eEF58AEfF1);
        } else if (chainId == 31337) {
            return address(0);
        } else {
            revert("Unsupported chainId");
        }
    }
```

The issue is that when relying on the WETH to USDT pool, and assuming the protocol correctly retrieves the price of 1 WETH in terms of USDT, USDT itself is not directly priced in USD. This means the protocol implicitly assumes that USDT has a 1 to 1 peg with USD, which is not always true.

As a result, ETH can be mispriced when the Chainlink ETH to USD oracle returns stale or incorrect values. One impact is that if USDT is priced slightly above USD, the `tokensToMint` minted to users through `LaunchManager` will be lower than intended. This mispricing also affects the tax calculation applied during swaps in the `OlympusXHook::_beforeSwap` flow.

```
    function getLatestEthPriceUSD() public view returns (uint256 price) {
        PriceConfig memory config = priceConfig;
@>        price = _getChainlinkPrice(config.chainlinkFeed, config.oracleStaleThreshold);

@>        if (price == 0) {
@>            return _getUniswapTWAP(config.uniswapFallback);
        }
    }
    // ...
    function _getUniswapTWAP(UniswapConfig memory config) internal view returns (uint256
price) {
        uint160 twapSqrtPriceX96;
        address tokenIn;
        address tokenOut;
        if (config.isUniV4) {
            PoolKey memory poolKey = abi.decode(config.poolData, (PoolKey));
            twapSqrtPriceX96 = V4OracleLib.getTwapSqrtPriceX96(poolKey, config.twapPeriod);
            tokenIn = Currency.unwrap(poolKey.currency0);
            tokenOut = Currency.unwrap(poolKey.currency1);
        } else {
            bytes32 poolAddress = bytes32(uint256(uint160(config.poolAddress)));
            twapSqrtPriceX96 = V3OracleLib.getTwapSqrtPriceX96(poolAddress,
config.twapPeriod);
            IUniswapV3Pool pool = IUniswapV3Pool(config.poolAddress);
            address token0 = pool.token0();
            address token1 = pool.token1();
            tokenIn = config.weth == token0 ? token0 : token1;
```

```
            tokenOut = config.weth == token0 ? token1 : token0;
        }

@>          price = V3OracleLib.getQuoteForSqrtRatioX96(twapSqrtPriceX96, 1e18, tokenIn,
tokenOut);
        uint8 outDecimals = IERC20Metadata(tokenOut).decimals();
        if (outDecimals != 18) {
            price = price * (10 ** (18 - outDecimals));
        }
    }
```

### Recommendations

The recommendation is to include an additional oracle that prices USDT in terms of USD so that ETH pricing remains accurate when relying on Uniswap-based fallback mechanisms.

## [M-05] Overflow risk in price calculation during V4 pool creation

### Severity

**Impact:** High

**Likelihood:** Low

### Description

It is observed that in `LaunchManager`, once `initialLiquidityEthCollected` reaches the required `initialLiquidityEth`, the protocol proceeds to initialize a V4 `ETH <> OLY` pool through the V4 `PoolManager` contract.

```
    function createAndInitializeLiquidityPool(uint48 deadline) external {
        require(initialLiquidityEthCollected >= initialLiquidityEth,
LaunchManager__InsufficientInitialEth());
        require(initialPoolTokenPrice > 0, LaunchManager__InvalidInitialPoolPrice());

        uint256 ethPrice = ethPriceFeed.getLatestEthPriceUSD();

        // Calculate liquidity pool allocation dynamically based on ETH price and target
token price
        uint256 liquidityPoolAllocation = (initialLiquidityEth * ethPrice) /
initialPoolTokenPrice;

        PoolKey memory key = PoolKey({
            currency0: CurrencyLibrary.ADDRESS_ZERO,
            currency1: Currency.wrap(address(olympusX)),
            fee: 1_0000,
            tickSpacing: 200,
            hooks: IHooks(address(olympusXHook))
        });
```

```
        uint160 sqrtPriceX96 = calculateSqrtPriceX96(initialLiquidityEth,
liquidityPoolAllocation);
        // ...
```

The protocol uses the spot price derived from `reserves(OLY) / reserves(initialLiquidityEth)` to compute the square root `Q64.96` price for pool initialization.

```
    function calculateSqrtPriceX96(uint256 _amount0, uint256 _amount1) public pure returns
(uint160 sqrtPriceX96) {
        // 1e48 is used for high precision
        sqrtPriceX96 = uint160((Math.sqrt((_amount1 * 1e48) / _amount0) << 96) / 1e24);
    }
```

However, there is an issue with the use of `1e48` to achieve higher precision. The maximum value that a `uint256` can store in Solidity is approximately `1.1579208924 * 10^77`, which is `115792089237316195423570985008687907853269984665640564039457584007913129639935`.

If we assume `initialEthRequired = 20 eth`, `initialPoolTokenPrice = 0.0000001 ether`, which is `$0.0000001 USD`, and the price of eth is `4000e18`, the square root price calculation involves a term equivalent to `8e29 * 1e48`.

```
    liquidityPoolAllocation = 20e18 * 4000e18 / 0.0000001e18 = 8e29
    sqrtPriceX96 = sqrt(8e29 * 1e48 / 20e18 * 2 ** 96 / 1e24)
```

This value evaluates to `8e77`, which exceeds the maximum `uint256` value that Solidity can store, which is `1.1579208924 * 10^77`. As a result, the calculation overflows.

The impact is that the V4 pool cannot be successfully launched due to this overflow during price initialization.

However, it is to be noted that this is of low likelihood with realistic ETH price and targeted `OLY` price value ( `0.0125e18`, $0.0125 USD in tests), and the chances of this happening are of low likelihood.

### Recommendations

Recommendation is to reduce the precision used in the calculation to a lower scale in order to avoid exceeding the maximum representable value.

## [M-06] Inflation pool accounting gap allows excess distribution

### Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The inflation pool accounting system has a timing gap between when stakes end and when inflation rewards are claimed that allows more inflation to be distributed than the `maxInflationPool` limit allows.

The system tracks emitted inflation using two variables:

- `totalInflationDistributed`: Inflation allocated to daily pools but not yet minted.
- `totalInflationMinted`: Inflation that has been minted to users.

The available inflation for distribution is calculated in `_distributeDailyInflation()` as:

```
uint256 maxAllowedTotal = maxInflationPool > totalInflationMinted ? maxInflationPool - totalInflationMinted : 0;

uint256 maxAdditional = maxAllowedTotal > totalInflationDistributed ? maxAllowedTotal - totalInflationDistributed : 0;
```

So basically `maxInflationPool - totalInflationMinted - totalInflationDistributed`.

When `stakeEnd()` or `goodAccounting()` processes a stake ending, the code reduces `totalInflationDistributed` by the full `accumulatedInflation` amount.

```
totalInflationDistributed -= accumulatedInflation;
```

but `totalInflationMinted` is not updated. The inflation is only unlocked for the user via `userUnlockedInflation`, which can be claimed later via `claimInflationRewards()`.

Only when the user calls `claimInflationRewards()` does the `totalInflationMinted` increase by the claimed amount. This creates a gap where:

1. Before stake end: `maxInflationPool = 1000`, `totalInflationMinted = 0`, `totalInflationDistributed = 1000`, `totalInflationMinted = 0` → available = 1000 - 0 - 1000 = 0
2. After stake end: `maxInflationPool = 1000`, `totalInflationDistributed = 0` (reduced by 1000), `totalInflationMinted = 0`
   → available = 1000 - 0 - 0 = 1000 (incorrectly)
3. After claim: `maxInflationPool = 1000`, `totalInflationDistributed = 0`, `totalInflationMinted = 1000`
   → available = 1000 - 1000 - 0 = 0

During the gap between steps 2 and 3, `_distributeDailyInflation()` can distribute additional inflation up to the full `maxInflationPool` amount, even though that inflation will eventually be minted when users claim their unlocked rewards. This allows the total inflation to exceed the intended `maxInflationPool` limit.

### Recommendation

Fix the accounting to properly track inflation through the unlock to claim lifecycle:

1. In `stakeEnd()` and `goodAccounting()`, only reduce `totalInflationDistributed` by `rewardPenalty` (the penalty amount that goes back to the inflation pool for redistribution), not by the full `accumulatedInflation`.
2. In `claimInflationRewards()`, reduce `totalInflationDistributed` by the claimed amount and increase `totalInflationMinted` accordingly.

## [M-07] Tokens permanently locked in accounting if auction receives no deposit

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The `OlympusXAuction` contract suffers from an accounting error where OLY tokens allocated to an auction cycle become permanently inaccessible if no users deposit ETH during that cycle. This occurs due to improper management of the `totalReservedTokens` state variable across the auction lifecycle.

When `startDistribution()` is called to begin a new auction cycle, the function calculates the amount of tokens to distribute and increments `totalReservedTokens`:

```
uint256 amountToDistribute = ((poolBalance - totalReservedTokens) *
uint256(distributionBps)) / BPS;
olympusXDistributedPerCycle[currentCycle] = amountToDistribute;
totalReservedTokens = totalReservedTokens + amountToDistribute;
```

These tokens are marked as "reserved" for that cycle's participants. The `totalReservedTokens` variable is only decremented when users successfully claim their tokens via `claim()`, which subtracts the claimed amount.

However, the `getClaimableAmount()` function contains an early return that prevents any claims if no ETH was deposited during the cycle.

This creates a deadlock scenario:

1. Tokens are allocated and added to totalReservedTokens when the cycle starts.
2. If no users deposit ETH during the 8-day auction period (due to market conditions, lack of awareness, or technical issues), totalEthDeposited[currentCycle] remains zero.
3. No user can claim tokens from that cycle because getClaimableAmount() returns zero.
4. The tokens remain physically in the contract but are forever counted as "reserved" in totalReservedTokens.
5. These tokens cannot be redistributed in future auctions.

### Recommendation

Implement a mechanism to release unreserved tokens from failed auctions when starting a new distribution cycle. This ensures that tokens from auctions with zero participation are released back into the available pool for future distributions.

## [M-08] Accrued fees can be exploited to obtain subsidized liquidity positions

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The `unlockCallbackPlace()` function in `OlympusXHook` allows users to place limit orders by adding liquidity to specific tick ranges. However, when the pool price enters a tick range but does not fully cross it (partial fill), the existing position accumulates trading fees. These accrued fees create an exploitable condition where new users can add liquidity to the same tick and effectively use the accumulated fees to subsidize their position costs.

When `poolManager.modifyLiquidity()` is called on a position with accrued fees, the returned `BalanceDelta` includes both the cost of adding new liquidity and the credit from accumulated fees. The function only checks the final delta values to ensure the position is out of range:

```
(BalanceDelta delta,) = poolManager.modifyLiquidity(
    key,
    ModifyLiquidityParams({
        tickLower: tickLower, tickUpper: tickLower + key.tickSpacing, liquidityDelta:
liquidityDelta, salt: 0
    }),
    ZERO_BYTES
);
```

```
if (delta.amount0() < 0) {
    if (delta.amount1() != 0) revert InRange();
    // ... settle amount0
}
```

The checks if `(delta.amount1() != 0) revert InRange();` are designed to ensure only out-of-range limit orders are placed. However, these checks operate on the net delta after fees are applied, not the raw costs. An attacker can manipulate the liquidity amount to create a scenario where accrued fees perfectly offset one token's requirement.

Attack Scenario:

1. Initial state: Tick range 124,000-124,200 has an existing limit order with 1 ETH of accrued trading fees.
2. Normal cost to add liquidity in this range would be 1 ETH + 3,000 USDC (assuming current price).
3. Since the accrued fees are subtracted from the token amounts required for adding liquidity, an attacker can add liquidity worth 6,000 USDC (1 ETH * 3,000 USDC + 3,000 USDC) by only paying 3,000 USDC.
4. The attacker adjusts their liquidity to target a net delta of 0 ETH, -X USDC.
5. The checks pass because delta.amount0() == 0 and delta.amount1() < 0.
6. The attacker pays only 3,000 USDC but receives a position worth 1 ETH + 3,000 OLY, effectively stealing the 1 ETH of accrued fees.

The vulnerability exists because the contract does not track or validate whether fees were consumed during the liquidity addition. The second return value from `modifyLiquidity()` contains `feesAccrued` information but is currently ignored.

This allows sophisticated users to monitor partially filled tick ranges and exploit accumulated fees, providing them with free liquidity at the expense of existing limit order providers who earned those fees.

### Recommendation

Check the feesAccrued return value from modifyLiquidity() and revert if any fees have been accrued, preventing liquidity additions to ticks with accumulated fees. This ensures that only clean tick ranges without accumulated fees can accept new limit orders, protecting fee earnings for existing liquidity providers.

## [M-09] New limit order participants unfairly receive fees from exited positions

### Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The `place()` function in `OlympusXHook` allows users to add liquidity to existing epochs that already have accumulated fees from previous participants who exited their positions. This creates an unfair distribution mechanism where new participants receive trading fees they never earned, diluting rewards for users who legitimately held positions while those fees were generated.

When a user removes their limit order via `kill()` while other liquidity providers remain in the same epoch, the exiting user's accumulated trading fees are allocated to remaining participants through the `epochInfo.token0Total` and `epochInfo.token1Total` variables:

```
function unlockCallbackKill(...) external selfOnly returns (...) {
    if (!removingAllLiquidity) {
        (, BalanceDelta deltaFee) = poolManager.modifyLiquidity(
            key,
            ModifyLiquidityParams({tickLower: tickLower, tickUpper: tickUpper,
liquidityDelta: 0, salt: 0}),
            ZERO_BYTES
        );
        if (deltaFee.amount0() > 0) {
            poolManager.mint(address(this), key.currency0.toId(), amount0Fee =
uint128(deltaFee.amount0()));
        }
        // ...
    }
    // Later in kill():
    epochInfo.token0Total += amount0Fee;
    epochInfo.token1Total += amount1Fee;
}
```

The problem arises because the `place()` function has no validation preventing new users from joining epochs with existing `token0Total` or `token1Total` balances. When new users add liquidity to such epochs, they immediately become entitled to a proportional share of those accumulated fees upon withdrawal, despite never having held the position while those fees were earned.

Attack Scenario:

1. User A and User B place limit orders in tick range 124,000-124,200, each depositing 1 ETH worth of liquidity.
2. Over several days, the price moves in and out of this range (without fully filling), accumulating trading fees in ETH and OLY.
3. User A decides to exit and calls `kill()`, receiving their principal back.
4. All accumulated fees are added to `epochInfo.token0Total` and `epochInfo.token1Total`, intended for User B as the remaining participant.
5. User C monitors the contract, sees the epoch has accumulated fees, and calls `place()` to add 1 ETH of liquidity (while the pool tick is outside of the tick range).
6. Now User B and User C each have equal liquidity in the epoch.
7. When the epoch fills and both withdraw, all accumulated fees are split equally between User B and User C.
8. User C received fees he never earned, effectively stealing half of User B's legitimate fee earnings.

The `place()` function loads the existing `epochInfo` but performs no validation on whether any fees have accumulated.

### Recommendation

Add a validation check in the `place()` function to prevent adding liquidity to epochs that already have accumulated fees. This ensures that accumulated fees are only distributed among participants who were present when those fees were earned, maintaining fair reward distribution.

## [M-10] Withdrawal requests revert when combined amounts exceed Lido's `MAX_WITHDRAWAL`

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The `_prepareWithdrawalAmounts()` function in `ValidatorVault` attempts to handle small remainder amounts by combining them with a full withdrawal request. However, this logic creates amounts that exceed Lido's `MAX_WITHDRAWAL_AMOUNT` of 1000 stETH, causing all withdrawal requests to revert and blocking the protocol's ability to harvest and distribute rewards.

Lido's withdrawal queue enforces strict constraints:

- Maximum per request: 1000 stETH (exactly 1000 ether).
- Minimum per request: 100 wei.

The function handles remainders smaller than the minimum by merging them with the last full request:

```
} else if (remainder > 0 && fullRequests > 0) {
    // If remainder is too small but we have full requests,
    // we'll adjust the last full request
    unchecked {
        fullRequests -= 1;
@>        remainder = remainder + MAX_WITHDRAWAL_AMOUNT;  //@audit: exceed max!
        totalRequests = fullRequests + 1;
    }
}
// Later:
if (remainder >= MIN_WITHDRAWAL_AMOUNT) {
    amounts[fullRequests] = remainder;  // This amount exceeds Lido's limit
}
```

Problematic Scenario:

1. Validator vault needs to withdraw 1000.00000000000000050 ether (1000 stETH + 50 wei).
2. Function calculates:
    - fullRequests = 1000 ether / 1000 ether = 1
    - remainder = 50 wei
3. Since remainder < MIN_WITHDRAWAL_AMOUNT (50 < 100 wei) and fullRequests > 0:
    - fullRequests = 0
    - remainder = 50 wei + 1000 ether = 1000.00000000000000050 ether
    - totalRequests = 1
4. Array constructed: amounts[0] = 1000.00000000000000050 ether.
5. When requestWithdrawals() is called on Lido's withdrawal queue, it reverts because 1000.00000000000000050 ether > 1000 ether MAX_WITHDRAWAL_AMOUNT.
6. The entire harvest() transaction fails.

## Recommendation

When the combined remainder would exceed `MAX_WITHDRAWAL_AMOUNT`, split it into two requests that both respect Lido's constraints. One with `amount = MAX_WITHDRAWAL_AMOUNT / 2` and a second with `amount2 = MAX_WITHDRAWAL_AMOUNT / 2 + remainder`. This ensures all withdrawal requests respect Lido's maximum while properly handling small remainders, preventing harvest failures.

# [M-11] Imprecise maturity calculation allows premature stake ending

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The `_verifyMaturityPercentage()` function enforces that stakes can only be ended early if they have reached at least 50% of their committed duration. However, the current implementation uses a day-based calculation that compounds rounding errors from both the stake start time and the current time, allowing users to end stakes up to nearly a full day before they have actually reached 50% maturity.

The flawed calculation occurs in `_verifyMaturityPercentage()`:

```
uint256 stakeStartDay = (_stake.stakeInfo.stakeStartTs - launchTime) / uint40(1 days) + 1;
uint256 daysCompleted = currentDay - stakeStartDay;
uint256 maturityPercent = (daysCompleted * 100) / _stake.stakeInfo.numOfDays;
```

Both stakeStartDay and currentDay are calculated by dividing the elapsed seconds by 1 day (86400 seconds), rounding down, then adding 1. This creates a scenario where timestamps up to 23 hours and 59 minutes apart can be counted as the same day.

Concrete example demonstrating the issue:

1. Protocol launches at timestamp 0 (launchTime).
2. User creates a 100-day stake at timestamp 86,399 (23:59:59 after launch)
    - stakeStartTs = 86,399
    - stakeStartDay = 86399 / 86400 + 1 = 0 + 1 = 1
3. User attempts early end at exactly 50 days + 1 second after the launch (timestamp: 86400 * 50 + 1 = 4,320,001)
    - Actual time staked: (4,320,001 - 86,399) / 86,400 = 49.000002 days
    - currentDay = 4,320,001 / 86,400 + 1 = 50 + 1 = 51
    - daysCompleted = 51 - 1 = 50
    - maturityPercent = (50 * 100) / 100 = 50%
4. The check passes (50% >= 50%) even though only 49 days have elapsed.

The user only staked for 49% of the time but is still able to end his stake early.

### Recommendation

Calculate maturity based on actual elapsed time using timestamps rather than protocol days:

```
function _verifyMaturityPercentage(UserStake storage _stake) internal view {
    // Calculate actual time elapsed since stake start
    uint256 timeElapsed = block.timestamp - _stake.stakeInfo.stakeStartTs;
    uint256 daysCompleted = timeElapsed / 1 days;
    uint256 maturityPercent = (daysCompleted * 100) / _stake.stakeInfo.numOfDays;
    require(maturityPercent * 100 >= MIN_MATURITY_BPS, StakingVault__StakeNotMatured());

}
```

This approach measures the actual duration the stake has been active rather than relying on the protocol's day counting system, ensuring the 50% maturity requirement is accurately enforced based on real elapsed time.

## [M-12] Incorrect validation in `validateFarmSetup` allows non-reward tokens in V4 farms

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The `validateFarmSetup()` function incorrectly validates which token must be a registered reward token for Uniswap V4 pools. In V4 pools, ETH is represented as `address(0)` in `currency0`, while `currency1` holds the paired ERC20 token. However, the validation logic attempts to determine the "other token" using the `isWethToken` mapping, which does not contain `address(0)` since address(0) cannot be added to it:

```
function setWethToken(address _wethToken, bool _isWethToken) external onlyOwner {
@>      require(_wethToken != address(0), FarmKeeper_InvalidAddress());
        isWethToken[_wethToken] = _isWethToken;
        emit FarmKeeper_WethTokenSet(_wethToken, _isWethToken);
}
```

This means that for a V4 ETH pool, the function checks currency0 (which is ETH), and since the result is false, determines ETH to be the other token instead of currency1:

```
if (isEthPool) {
    address otherToken = isWethToken[currency0] ? currency1 : currency0;
    if (!IStakingVault(stakingVault).isRewardToken(otherToken)) {
        revert FarmKeeper_TokenNotRegistered();
    }
}
```

This means the validation checks whether ETH is a reward token rather than validating currency1, allowing farms to be created with currency1 tokens that are not registered as reward tokens.

This breaks a critical security guarantee: all non-ETH/WETH farm tokens must be registered reward tokens to enable proper fee distribution. When such a misconfigured farm attempts to distribute fees through harvestFarm() or removeFarm(), the transaction will revert. The distributeRewards() function calls distribute28DayRewards() for currency1, which internally validates isRewardToken[_token] in StakingVault._distributeRewards() and reverts with "StakingVault__InvalidToken()" if the token is not registered.

This renders the farm permanently unable to harvest or be removed, effectively locking all accumulated fees and liquidity in the farm. The farm will continue collecting fees but cannot distribute them to stakers, and the owner cannot remove the farm to recover funds.

### Recommendation

For V4 pools, if the pool isEthPool, currency0 is always ETH (address(0)) by design, so the validation should directly check currency1 without attempting to determine which is the "other token".

## [M-13] Missing validation allows intermediary pool with wrong token causing fund loss

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

When enabling a farm with a non-native pool (neither token is ETH/WETH), an intermediary pool is required to swap ETH into one of the main pool's tokens. However, `validateFarmSetup()` does not validate that the non-ETH/WETH token in the intermediary pool matches either `currency0` or `currency1` of the main pool. The validation only checks:

1. Both main pool tokens are registered reward tokens.
2. The intermediary pool exists and has proper TWAP history.
3. Exactly one token in the intermediary pool is ETH/WETH.

But crucially, it never validates that the other intermediary pool token is actually one of the main pool tokens.

This allows an owner to configure an intermediary pool with an arbitrary token X, where X is neither the main pool's `currency0` nor `currency1`. When ETH is swapped through this misconfigured intermediary pool:

```
// In FarmKeeperV3Lib.processEthSwapResult()
if (Currency.unwrap(farm.poolKey.currency0) == intermediaryToken) {
    farm.currency0Balance = farm.currency0Balance + swapResult.amountOut;
} else {
    farm.currency1Balance = farm.currency1Balance + swapResult.amountOut;
}
```

The code assumes the intermediary token is either `currency0` or `currency1`. If it is not `currency0`, the else branch incorrectly adds the swapped amount to `farm.currency1Balance`, even though the contract actually received token X. This creates a critical accounting mismatch where the farm's balance tracking indicates it holds `currency1` tokens when it actually holds token X.

When `depositToFarm()` subsequently attempts to add liquidity to the main pool, it will try to use `farm.currency0Balance` and `farm.currency1Balance` with the main pool's `currency0` and `currency1` tokens. The Uniswap PositionManager will attempt to pull `currency1` tokens from the FarmKeeper contract, but the contract does not have `currency1` - it has token X. The transaction will revert with insufficient balance, and token X remains permanently stuck in the FarmKeeper with no recovery mechanism. All subsequent ETH allocations to this farm will continue swapping to token X and getting stuck, effectively creating a black hole for allocated funds.

### Recommendation

Add validation to ensure the non-ETH/WETH token in the intermediary pool matches one of the main pool tokens.

## [M-14] Reward debt heuristic can drop rewards after reward token readdition

### Severity

**Impact:** High

**Likelihood:** Low

### Description

`StakingVault` uses a heuristic where `userRewardDebt == 0` is treated as "uninitialized" and replaced with `tokenAdditionBaseline`. In a `removeRewardToken(token) → addRewardToken(token)` scenario, this can cause loss of claimable rewards for some users: a user who staked when `cumulativeRewardPerShare[token][cycle] == 0` and never claimed and never changed shares will have `userRewardDebt` remain `0`. After the token is

re-added, `tokenAdditionBaseline` is reset to a newer (non-zero) cumulative value, and during claim the `userDebt == 0` branch substitutes this new baseline, effectively skipping rewards accrued before the re-add.

**Baseline is reset on token addition**

```
tokenAdditionBaseline[_token][CYCLE_8_DAYS] = cumulativeRewardPerShare[_token][CYCLE_8_DAYS];
...
```

**Debt can be initialized to zero**

```
userRewardDebt[_user][token][CYCLE_8_DAYS] = cumulativeRewardPerShare[token][CYCLE_8_DAYS];
...
```

**Critical substitution: `userRewardDebt == 0` is replaced with the (new) baseline**

```
if (userDebt == 0) {
    userDebt = tokenAdditionBaseline[_token][_cycleNo];
}
```

Because `userDebt == 0` is ambiguous ("never initialized" vs "initialized to 0"), users matching the described conditions can permanently lose the ability to claim rewards accrued before the token was removed and re-added.

### Recommendations

Do not use `userDebt == 0` as an indicator of "uninitialized" debt. Instead:

- store an explicit initialization flag (e.g., `userDebtInitialized[user][token][cycle]`) and apply `tokenAdditionBaseline` only when the debt is not initialized, not when it equals zero; or
- introduce a "generation/version" for reward tokens (incremented on add/remove) and store `userDebtGen` to distinguish old debts from a new token period.

## [M-15] Excess ETH not refunded when placing limit orders

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The `place()` function in `OlympusXHook` accepts ETH payments via `msg.value` to place limit orders, but fails to refund any excess ETH that exceeds the amount actually required for the liquidity position. This results in users losing funds when they send more ETH than necessary, with the excess permanently trapped in the contract.

When a user calls `place()` to create a limit order with ETH as `currency0`, the function calculates the exact amount needed through the `unlockCallbackPlace()` callback.

Inside `unlockCallbackPlace()`, when `delta.amount0() < 0` (indicating ETH is needed), the function settles with only the required amount:

```
amount0 = uint256(uint128(-delta.amount0()));

poolManager.sync(key.currency0);

poolManager.settle{value: amount0}();
```

The critical issue is that the `place()` function never compares `msg.value` with the actual `amount0` used, nor does it return any excess. If a user sends 1 ETH but only 0.5 ETH is needed for the liquidity position, the remaining 0.5 ETH becomes stuck in the contract.

The OlympusXHook contract has no administrative function to recover trapped ETH. Once excess ETH is sent and not used for the limit order, it becomes permanently inaccessible.

### Recommendation

Implement a refund mechanism in the `place()` function to return any unused ETH to the caller. This can be as simple as checking the current ETH balance of the hook and returning all ETH to the caller.

## [M-16] Unequal reward distribution lets short-term stakers access long-term rewards

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The protocol distributes rewards across five different cycles (8, 28, 90, 369, and 888 days), intending to reward longer-term stakers with access to less frequent but potentially more valuable reward cycles. However, the current implementation fails to restrict which cycles a stake is eligible for based on its duration, leading to a significant fairness issue where short-term stakers can receive the same rewards as long-term stakers.

When a user creates a stake, their shares are calculated via `StakingCalculations.calculateShares()` which applies a time-based bonus (0-400% depending on stake duration from 88 to 1776 days). However, once shares are minted, they

participate in **all five reward cycles** regardless of the stake's duration. The protocol tracks only the global `totalShares` variable and uses it uniformly across all cycles when distributing rewards in `_triggerCyclePayout()`:

```
cumulativeRewardPerShare[token][_cycleNo] += (tokenReward * PRECISION) / _globalActiveShares;
```

This creates an exploitable scenario where a user who stakes for the minimum duration of 88 days receives identical rewards to a user who commits to the maximum duration of 1,776 days, assuming they have equal shares.

Example scenario:

1. User A stakes 1000 tokens on day 1 for 888 days, receiving 1000 shares (hypothetically, for simplicity, the time-based bonus is ignored).
2. User B stakes 1000 tokens on day 800 for 88 days, receiving 1000 shares.
3. Both stakes mature around day 888.
4. When the 888-day cycle payout triggers on day 888, both users receive equal rewards from that cycle despite User B only committing to 88 days.
5. User B gets to unlock and exit their position at the same time as User A while having earned the same 888-day cycle rewards as User A, who locked for 10x longer.

The issue is exacerbated because the time-based bonus on shares does not correlate with cycle eligibility. A sophisticated user could stake for shorter durations timed to mature just before major cycle payouts, earning outsized rewards relative to their commitment without facing the opportunity cost of long-term locking.

## Recommendation

Implement cycle-specific share tracking that enforces eligibility requirements. Each stake should only participate in reward cycles that align with or are shorter than its committed duration. For example:

- 88-day stakes: eligible for 8-day and 28-day cycles only.
- 888-day stakes: eligible for all cycles (8, 28, 90, 369, and 888).

The recommended implementation approach:

1. Add cycle-specific share tracking:
2. When starting a stake, determine which cycles the stake qualifies for based on duration and add shares to those specific cycles.
3. Store stake duration in the UserStake struct and track which cycles each stake participates in.
4. When ending a stake, remove shares from the appropriate cycle pools.
5. Update _triggerCyclePayout() to use cycle-specific share counts.
6. Update _claimUserTokenCycleReward() to only allow claiming from cycles the user’s stakes are eligible for.

This ensures that longer-term commitments are properly rewarded while maintaining fairness across different staking durations.

## [M-17] Inflation reward calculation is not accurate

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

In StakingVault, users can stake OLY to earn some staking rewards.

The inflation rewards will be calculated by the formula `totalSupply * annualRateBPS * delta time`.

Once users stake or unstake their positions, the previous time slot's inflation rewards will be calculated by `_distributeDailyInflation`.

The problem here is that the `totalSupply` will change without any stake. For example, users mint OLY and do not stake any OLY into the staking vault, or users can burn OLY via BuyAndBurn.

For example, users mint OLY and do not stake any OLY into the staking vault, or users can burn OLY via

1. In timestamp X, users stake OLY into the staking vault.
2. In timestamp X + 3000, users mint some OLY and stake no OLY into the staking vault. The `totalSupply` for OLY will increase from timestamp X + 100.
3. In timestamp X + 4000, users want to stake OLY into the staking vault. The inflation rewards between timestamp X and timestamp X + 4000 will be calculated based on the latest `totalSupply`. Users may gain more rewards than expected.

```
    function unlockCallbackKill(
        PoolKey calldata key,
        int24 tickLower,
        int256 liquidityDelta,
        address to,
        bool removingAllLiquidity
    ) external selfOnly returns (uint128 amount0Fee, uint128 amount1Fee, uint256 amount0,
uint256 amount1) {
        int24 tickUpper = tickLower + key.tickSpacing;
        if (!removingAllLiquidity) {
            (, BalanceDelta deltaFee) = poolManager.modifyLiquidity(
                key,
                ModifyLiquidityParams({tickLower: tickLower, tickUpper: tickUpper,
liquidityDelta: 0, salt: 0}),
                ZERO_BYTES
```

```
            );

            if (deltaFee.amount0() > 0) {
                poolManager.mint(address(this), key.currency0.toId(), amount0Fee =
uint128(deltaFee.amount0()));
            }
            if (deltaFee.amount1() > 0) {
                poolManager.mint(address(this), key.currency1.toId(), amount1Fee =
uint128(deltaFee.amount1()));
            }
        }
    }
```

### Recommendations

When OLY tokens are minted or burned, calculate the pending inflation rewards in a timely manner.

## [M-18] Ethereum can be locked if required liquidity is not achieved

### Severity

**Impact:** High

**Likelihood:** Low

### Description

We observe that users supply ETH to mint `OLY` tokens, which is tracked through `initialLiquidityEthCollected` in `LaunchManager`. Once the supplied ETH reaches the `initialLiquidityEth` threshold required to seed the V4 ETH <> OLY pool, anyone can call `LaunchManager::createAndInitializeLiquidityPool()` to initialize the pool and add liquidity through the `FarmKeeper`.

However, if `initialLiquidityEthCollected` never reaches `initialLiquidityEth`, the supplied ETH remains permanently locked in `LaunchManager` with no mechanism for recovery.

Extending the mint phase does not guarantee that the `initialLiquidityEth` target will be met. In such a case, the only way forward would be for the protocol itself to supply ETH and mint `OLY` so that sufficient ETH can later be provided to the V4 pool through the `FarmKeeper`.

```
    function createAndInitializeLiquidityPool(uint48 deadline) external {
@>      require(initialLiquidityEthCollected >= initialLiquidityEth,
LaunchManager__InsufficientInitialEth());
        require(initialPoolTokenPrice > 0, LaunchManager__InvalidInitialPoolPrice());

        uint256 ethPrice = ethPriceFeed.getLatestEthPriceUSD();
```

```
        // Calculate liquidity pool allocation dynamically based on ETH price and target
token price
        uint256 liquidityPoolAllocation = (initialLiquidityEth * ethPrice) /
initialPoolTokenPrice;

        PoolKey memory key = PoolKey({
            currency0: CurrencyLibrary.ADDRESS_ZERO,
            currency1: Currency.wrap(address(olympusX)),
            fee: 1_0000,
            tickSpacing: 200,
            hooks: IHooks(address(olympusXHook))
        });
        // ...
```

### Recommendations

The recommendation is to introduce a recovery mechanism that allows ETH to be retrieved in scenarios where the required liquidity to seed the V4 pool is never achieved.

## [M-19] Emergency withdrawal fails to recover ETH in `pendingEth` leaving funds in `V`

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The `emergencyWithdrawWstEth()` function is designed to recover all funds from the `ValidatorVault` in extreme circumstances, but it only withdraws wstETH tokens and completely ignores any ETH held in the `pendingEth` state variable. This creates a scenario where a portion of user funds is left in the contract after an emergency withdrawal, defeating the purpose of the emergency mechanism.

**The Issue:**

The emergency withdrawal function only handles wstETH:

```
function emergencyWithdrawWstEth() external onlyOwner {
    uint256 balance = IERC20(wstEth).balanceOf(address(this));
    if (balance == 0) revert ValidatorVault__ZeroAmount();

    IERC20(wstEth).approve(stakingVault, balance);
    IStakingVault(stakingVault).distributeRewards(wstEth, balance);
    emit EmergencyWithdraw(balance);
}
```

The `pendingEth` variable tracks ETH that has been received by the contract (via the `receive()` function) but not yet staked through Lido. This ETH sits in the contract's balance waiting for someone to call `stakeETH()`. If the owner wants to withdraw all wstETH, he should not be forced to stake the `pendingETH` first.

### Recommendation

Modify the emergency withdrawal function to include pending ETH, sending it to the StakingVault for proper distribution to stakers. Also, update the EmergencyWithdraw event to track both asset types.

## [M-20] Removed tokens lead to unclaimable rewards causing permanent loss of user funds

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

When a reward token is removed from the `StakingVault` via `removeRewardToken()`, users lose the ability to claim any pending rewards for that token through the standard claiming mechanisms like `claimAllRewards()` and the automatic reward claim when starting or ending a stake. The `_claimAllTokensForCycle()` function only iterates through the current `rewardTokens` array, meaning removed tokens are completely skipped.

**Root Cause:**

The `_claimAllTokensForCycle()` function only processes tokens currently in the `rewardTokens` array:

```
function _claimAllTokensForCycle(address _user, uint256 _cycleNo) internal returns (uint256
totalClaimed) {
    uint256 tokensLength = rewardTokens.length;

    for (uint256 t = 0; t < tokensLength; t++) {
        address token = rewardTokens[t]; // Only iterates current tokens
        uint256 tokenReward = _claimUserTokenCycleReward(_user, token, _cycleNo);
        // ... transfer logic
    }
}
```

When `removeRewardToken()` is called, it removes the token from the `rewardTokens` array, only allowing the claim of the outstanding rewards by calling `claimRewards()` and knowing the token address of the removed token.

This function `_claimAllTokensForCycle` is called in multiple critical scenarios:

1. claimAllRewards(): Users explicitly calling to claim all their rewards will not receive rewards for removed tokens.
2. _claimAllRewardsInternal() (line 1100): Called internally in two important scenarios:
3. When ending a stake: Users expect all rewards to be claimed when they end their stake.
4. When starting a new stake: If a user has existing shares, all rewards are auto-claimed.

The issue is compounded by `_claimUserTokenCycleReward():`

if (userShares == 0) return 0;

Even if a user discovers they have unclaimed removed token rewards and tries to call claimRewards(removedToken) after ending all their stakes (shares = 0), the function returns 0 rewards. The user has no way to recover their earned rewards.

Additional Context:

While `claimRewards(address _token)` allows claiming specific tokens, this only works if:

1. Users know the token was removed.
2. Users have non-zero shares.
3. Users manually call claimRewards() with the specific removed token address.

Most users will rely on claimAllRewards() or stake ending to auto-claim all rewards, both of which fail silently for removed tokens.

### Recommendation

Implement a tracking mechanism for removed reward tokens and provide a dedicated claiming function which can be called directly and which is called when starting and ending a stake. This ensures users will always receive their earned rewards, even after tokens are removed as reward tokens.

## [M-21] Missing slippage protection in `mint` function exposes users to token loss

### Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The `mint()` function lacks slippage protection when converting ETH to OLY tokens, breaking the protocol's guarantee that users receive the expected token amount for their ETH contribution. Users have no control over the minimum tokens they receive, exposing them to potentially significant losses due to ETH price fluctuations between transaction submission and execution.

When a user calls `mint()` with ETH, the function calculates the OLY tokens to mint using `tokensToMint = msg.value * ethPriceFeed.getLatestEthPriceUSD() / schedule.tokenPrice`. The `ethPriceFeed.getLatestEthPriceUSD()` function queries a Chainlink oracle (or Uniswap TWAP fallback) to get the current ETH price in USD. However, this price can change between when the user submits their transaction and when it is mined on-chain.

Consider this scenario: A user wants to mint OLY tokens when ETH is trading at $3,000. They submit a transaction to mint with 10 ETH, expecting to receive a specific amount of OLY tokens based on this price. However, due to network congestion or a sudden market movement, their transaction is only mined 30 minutes later when ETH has dropped to $2,700 (a 10% decline). The user now receives 10% fewer OLY tokens than they expected, with no way to prevent or revert the transaction.

The issue is exacerbated by several factors:

1. **High ETH volatility**: ETH can experience significant price swings within minutes, especially during market volatility or major news events.
2. **Network congestion**: During periods of high gas prices, transactions may sit in the mempool for extended periods, increasing exposure to price changes.
3. **No minimum output parameter**: Unlike standard DEX interactions that require `minAmountOut`, users cannot specify acceptable slippage bounds.
4. **Irreversible loss**: Once executed, the transaction cannot be reverted, and users receive fewer tokens with no recourse.

While the impact is limited to the individual transaction (users do not lose their entire ETH, just receive fewer tokens than expected), this occurs with a high likelihood given ETH's natural price volatility. For large mints during volatile market conditions, the dollar value of lost tokens can be substantial.

## Recommendation

Add a `minTokensOut` parameter to the `mint()` function that allows users to specify the minimum acceptable token amount. Additionally, consider implementing a helper view function that allows users to preview the expected token amount before submitting their transaction, enabling them to set appropriate `minTokensOut` values with a chosen slippage tolerance (e.g., 1-2 percent).

# [M-22] Dead stake violates zero staker invariant and absorbs inflation share

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

`StakingVault` initializes a permanent `deadStake` with `DEAD_SHARES = 1000`, which is always included in `totalShares` and participates in inflation distribution. This has two related effects that stem from the same design choice.

First, the protective guard intended to prevent inflation accrual when no real stakers exist ( `if (_totalShares == 0) ... return` ) never triggers in practice. Because `_totalShares` always includes `DEAD_SHARES`, inflation can be distributed and reserved even when there are no real stakers in the system, violating the invariant "no stakers → no inflation accrual".

Second, inflation is calculated based on `totalSupply()` but distributed proportionally across `totalShares`, including the dead shares. As a result, a portion of every inflation distribution is allocated to the dead stake. Since the dead stake has status `BURNED` and cannot go through `stakeEnd()`, this portion of inflation is never released from accounting and remains permanently included in `totalInflationDistributed`.

This leads to sustained economic and accounting distortion:

- inflation can be distributed even in the absence of real stakers;
- part of the inflation pool is permanently absorbed by the dead stake without any beneficiary;
- the dead stake's share continuously increases `totalInflationDistributed`, applying pressure to the inflation cap and potentially reducing inflation available to real stakers.

The effect is most pronounced when `totalShares` is close to `DEAD_SHARES` (e.g., early protocol stages or low staking participation), while the inflation base ( `totalSupply()` ) may already be significant.

This is not an external exploit and does not result in direct loss of user funds, but represents a persistent deviation from the intended economic and accounting model of inflation distribution.

### Recommendations

**Recommendation A (minimal — restore invariant):**
Treat the state "no real stakers" as `totalShares == DEAD_SHARES` and handle it identically to the zero-shares case:

- do not distribute or reserve inflation;
- synchronize `lastInflationDistributionDay = _currentDay` and return.

**Recommendation B (stronger — exclude dead shares from inflation):**
Exclude dead shares from inflation distribution by computing:

```
effectiveShares = totalShares - DEAD_SHARES
```

and distributing inflation only across `effectiveShares`.
If `effectiveShares` == 0, perform the same early return as in Recommendation A.

This prevents the dead stake from absorbing inflation and avoids permanent inflation obligations that count against the cap without benefiting real stakers.

## [M-23] LP fees can be permanently locked in nonfilled epochs after all users call

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

In non-filled epochs, LP fees can accumulate in `epochInfo.token0Total` and `epochInfo.token1Total` as a result of `kill()` calls by non-last participants. When all users exit an epoch via `kill()` before it ever reaches the `filled` state, the epoch remains non-filled ( `filled == false` ), `liquidityTotal` drops to zero, and `withdraw()` becomes permanently unavailable due to the `NotFilled()` gate.

In this terminal state, accumulated token totals (representing collected LP fees) have no execution path for distribution. The epoch is neither finalized nor reset, and the recorded `token0Total` / `token1Total` values remain stranded on the hook without any user-accessible withdrawal mechanism.

`kill()` **reduces liquidity and adds fees without finalizing the epoch**

```
epochInfo.liquidityTotal -= liquidity;

epochInfo.token0Total += amount0Fee;
epochInfo.token1Total += amount1Fee;
```

**Fee collection for non-last participant in `unlockCallbackKill`**

```
if (!removingAllLiquidity) {
    (, BalanceDelta deltaFee) = poolManager.modifyLiquidity(...);

    if (deltaFee.amount0() > 0) {
        poolManager.mint(address(this), key.currency0.toId(), uint128(deltaFee.amount0()));
    }
    if (deltaFee.amount1() > 0) {
        poolManager.mint(address(this), key.currency1.toId(), uint128(deltaFee.amount1()));
    }
}
```

**Withdraw is blocked for non-filled epochs**

```
if (!epochInfo.filled) revert NotFilled();
```

There is no logic in `kill()` to finalize the epoch when `liquidityTotal` reaches zero (e.g., no forced fill, no reset, no distribution of accumulated `tokenTotals` to the last exiting participant). As a result, LP fees can remain permanently locked under a valid user flow where all participants cancel their positions before the epoch is filled.

### Recommendations

Avoid accumulating LP fees into `epochInfo.token0Total` / `epochInfo.token1Total` for non-filled epochs, or introduce explicit finalization logic when `liquidityTotal` reaches zero (e.g., distribute accumulated fees to the last exiting participant or reset the epoch state), ensuring that no token totals remain stranded in a non-filled epoch.

## [M-24] Missing collect accrued swap fees in `unlockCallbackPlace`

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

In OlympusXHook, users can place one limit order via `place()`. In the callback `unlockCallbackPlace`, the liquidity will be added to the pool.

The return value `BalanceDelta delta` includes needed token0/token1 and also accrued fees. It is possible that when users place one limit order, there are some accrued fees. This will cause users to fail to place a limit order because there is one condition in the function `unlockCallbackPlace`: the other token amount must be 0.

For example:

1. Place one limit order [tick A, tick A + tick spacing).
2. The current tick is moved to this range and accrues some fees for this position.
3. The tick moves out of this range.
4. If another user wants to place one limit order, this placement will be reverted.

```
    function unlockCallbackPlace(
        PoolKey calldata key,
        int24 tickLower,
        bool zeroForOne,
        int256 liquidityDelta,
        address owner
    ) external selfOnly returns (uint256 amount0, uint256 amount1) {
        // Add liquidity.
@>        (BalanceDelta delta,) = poolManager.modifyLiquidity(
            key,
            // tickLower, tickLower + tickSpacing.
            ModifyLiquidityParams({
                tickLower: tickLower, tickUpper: tickLower + key.tickSpacing,
liquidityDelta: liquidityDelta, salt: 0
            }),
            ZERO_BYTES
        );
        if (delta.amount0() < 0) {
@>            if (delta.amount1() != 0) revert InRange();
            if (!zeroForOne) revert CrossedRange();
            // For native ETH (currency0), settle with ETH
            amount0 = uint256(uint128(-delta.amount0()));
            poolManager.sync(key.currency0);
            poolManager.settle{value: amount0}();
        } else {
@>            if (delta.amount0() != 0) revert InRange();
            if (zeroForOne) revert CrossedRange();
            amount1 = uint256(uint128(-delta.amount1()));
            key.currency1.settle(poolManager, owner, amount1, false);
        }
    }
```

### Recommendations

Collect fees before adding liquidity.

## [M-25] Protocol rebalances incur `OLY` sell tax when swapping via taxed v4 farm pool

### Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

When rebalancing a v4 farm in `depositToFarm()`, `FarmKeeper` performs swaps **through the farm's own** `poolKey`. If the farm uses the **taxed main OLY pool (with** `OlympusXHook`**)**, then on the sell direction (OLY → ETH) the protocol effectively pays the **sell-tax** like a regular trader. With typical configurations where early-tier `taxBps` (e.g., 18–38%) exceeds the farm's default `swapSlippage` (e.g., 10%), sell-side rebalances become either operationally fragile (frequent `TooLittleReceived` / slippage reverts) or economically inefficient if operators increase slippage to force execution. This creates a systemic operational/economic degradation for farms that rebalance via the taxed pool: either sell rebalancing stalls, or it executes with predictable negative PnL (tax + pool fee + price impact), and higher allowed slippage increases execution/MEV risk for keeper-driven swaps. **Rebalance swaps are executed via the farm's** `poolKey`.

```
bool swapToken0 = farm.currency0Balance > farm.currency1Balance;
Currency swapTokenIn = swapToken0 ? farm.poolKey.currency0 : farm.poolKey.currency1;
Currency swapTokenOut = swapToken0 ? farm.poolKey.currency1 : farm.poolKey.currency0;
_depositToV4Farm(..., swapTokenIn, swapTokenOut, swapAmount, deadline);
```

**Swap execution relies on TWAP + configurable slippage**

```
SwapActions.SwapExactInParams({
amountIn: amountIn,
slippage: farm.swapSlippage,
twapPeriod: farm.twapPeriod
});
```

**Sell direction triggers OlympusXHook tax**
When the swap direction corresponds to selling OLY for the other currency in the taxed pool (as enforced by the hook's sell-tax rules), the protocol's rebalance swap is taxed, which directly reduces `amountOut` and increases the chance of failing the post-swap minimum out check.

## Recommendations

- Add an explicit hook exception for protocol-initiated rebalances (e.g., whitelist `FarmKeeper`) if consistent with the intended tokenomics; or
- Avoid protocol-initiated rebalances through the taxed pool by routing swaps through an untaxed intermediary path (for protocol operations), while keeping user trades taxed.

# Low findings

## [L-01] Stake position flooding risk

### Description

It is observed that a user stake position can be flooded with many small stake entries. An attacker can use the `LaunchManager::mint()` followed by `StakingVault::startStakeForUser()` flow to front run a user staking transaction and fill the staker `userStakes` array with numerous small stake positions. As a result, when the legitimate staker later tries to unstake the original stake position, they must remove each `stakeId` individually. Otherwise, the call sequence from `StakingVault::stakeEnd()` to `StakingVault::_processStakeEnd()` to `StakingVault::_removeUserStake()` may revert due to excessive gas consumption.

This situation can cause the legitimate staker to lose both the principal amount of `OLY` staked and the associated inflation rewards. The risk is higher if penalties apply after the stake passes the maturity grace period, especially when the `userStakes` array has been flooded with a very large number of entries and the user begins unstaking close to the end of the grace period.

The recommendation is to enforce an upper limit on the number of stake positions that can be stored in the `userStakes` array.

```
    function _removeUserStake(address _user, uint256 _stakeId) internal {
        uint256[] storage userStakes = userStakeIds[_user];
        uint256 length = userStakes.length;
        for (uint256 i = 0; i < length; i++) {
            if (userStakes[i] == _stakeId) {
                userStakes[i] = userStakes[length - 1];
                userStakes.pop();
                break;
            }
        }
    }
```

```
    function mint(address to, address referrer, uint256 vestingSchedule) external payable
override nonReentrant {
        require(vestingSchedule < _vestingSchedules.length,
LaunchManager__InvalidVestingSchedule());

        VestingSchedule storage schedule = _vestingSchedules[vestingSchedule];

        require(mintPhaseEndTimestamp > 0 && block.timestamp < mintPhaseEndTimestamp,
LaunchManager__NotInMintPhase());

        uint256 tokensToMint = msg.value * ethPriceFeed.getLatestEthPriceUSD() /
schedule.tokenPrice;

        require(tokensToMint <= schedule.totalTokens,
```

```
LaunchManager__VestingScheduleTotalTokensExceeded());

        schedule.totalTokens -= tokensToMint;
        // ...

@>        StakingVault(stakingVault).startStakeForUser(to, amounts, stakeDays);

        // ...
    }
```

## [L-02] Ineffective staking allowance management in `LaunchManager` due to hardcoding

### Description

`LaunchManager.setStakingVault()` attempts to manage staking permissions by revoking and re-approving token allowance ( `approve(0)` → `approve(type(uint256).max)` ). However, in `OlympusX` , the `allowance()` function special-cases the `stakingVault` address and always returns `type(uint256).max` , ignoring the stored allowance value.

As a result, the approve/revoke logic in `setStakingVault()` has no real effect on staking permissions. This creates a mismatch between the intended control flow in `LaunchManager` and the actual permission model enforced by `OlympusX` , potentially misleading maintainers, reviewers, or integrators into assuming that staking access can be revoked or rotated via allowance management.

### Recommendations

Either remove the ineffective `approve()` calls from `setStakingVault()` to avoid misleading logic, or refactor the permission model to rely on standard ERC20 allowance semantics if revocation/rotation of staking access is intended to be supported.

## [L-03] Claimable amount underflow causing stuck token

### Description

`OlympusXAuction` calculates the claimable amount of `OLY` tokens based on the amount of ETH used by a bidder through `OlympusXAuction::deposit()` . However, if we look at `OlympusXAuction::getClaimableAmount()` , the calculated claimable amount can underflow when `userDeposit * olympusXAmount < totalEth` for a given cycle. Although this scenario has a low likelihood, since ETH is expected to be worth more than the token, it can still occur.

When this underflow happens, `totalReservedTokens` is not decremented correctly. As a result, some `OLY` tokens can become stuck in the contract and cannot be distributed to future cycles. This leads to a situation where winning bidders, meaning those who deposited more ETH, do not receive the full amount of `OLY` tokens they should receive, while the remaining tokens are locked and unusable.

The suggestion is to introduce a redistribution mechanism that also properly decreases the `totalReservedTokens` variable so that any undistributed tokens can be handled correctly and made available for subsequent cycles.

```
    function getClaimableAmount(address user, uint256 cycle) public view override returns
(uint256) {
        uint256 userDeposit = userDeposits[user][cycle];
        if (userDeposit == 0) return 0;

        uint256 totalEth = totalEthDeposited[cycle];
        if (totalEth == 0) return 0;

        uint256 olympusXAmount = olympusXDistributedPerCycle[cycle];
        if (olympusXAmount == 0) return 0;

        // Calculate proportional share
@>        return (userDeposit * olympusXAmount) / totalEth;
    }
```

## [L-04] `EthPrice` oracle missing price deviation check

### Description

We observe that the `EthPrice` oracle uses either a UniswapV3 pool or a UniswapV4 pool as a fallback to price ETH in terms of USD. The primary pricing mechanism relies on the ETH USD price feed from Chainlink. However, we observe that there are no on-chain price deviation checks in place to handle scenarios where price spikes occur.

If the price of ETH temporarily spikes due to Chainlink oracles returning incorrect values or due to users swapping large amounts of tokens in the pools, although this is unlikely since high liquidity pools will be used and a very large amount of tokens would be required, the oracle may return an abnormally high ETH price in USD terms. One impact of this is that the tax applied on swaps could become extremely low, or the number of tokens minted to users in `LaunchManager` could become abnormally large for a very small amount of ETH.

The recommendation is to include on-chain price deviation checks in `EthPrice`.

```
    function getLatestEthPriceUSD() public view returns (uint256 price) {
        PriceConfig memory config = priceConfig;
@>        price = _getChainlinkPrice(config.chainlinkFeed, config.oracleStaleThreshold);

        if (price == 0) {
@>            return _getUniswapTWAP(config.uniswapFallback);
        }
    }
```

## [L-05] Mint transactions vulnerable to griefing via front running due to no partials

### Description

`LaunchManager.mint()` reverts if the calculated number of tokens to mint exceeds the remaining amount in the current vesting schedule. Because partial fills are not supported and the check is enforced as a hard `require`, users cannot buy "whatever remains" in the schedule, nor can they specify protective parameters such as minimum acceptable output.

This behavior enables a griefing/MEV scenario where a third party front-runs a large purchase with one or more small mint transactions, slightly reducing the remaining token amount. As a result, the original transaction—calculated against the previous remaining amount—reverts. The attacker gains no direct economic benefit, but can repeatedly force reverts and degrade availability, especially near the end of a vesting schedule where "buy-the-remainder" transactions are common.

### Recommendations

Consider supporting partial fills (minting the remaining tokens and refunding excess ETH) or adding user-defined protection parameters such as `minTokensOut` to reduce front-running griefing and improve mint availability near schedule exhaustion.

## [L-06] `ValidatorVault.harvest()` reverts on buy and burn rewards below Lido minimum

### Description

`ValidatorVault.harvest()` calculates 16% of rewards for Buy & Burn and attempts to create a Lido withdrawal request. Lido requires a minimum withdrawal amount of 100 wei stETH. If the Buy & Burn portion after conversion from wstETH is below this minimum, the function reverts with `ValidatorVault__AmountTooSmall()`, blocking the entire harvest process.

While rewards will eventually grow enough to overcome the minimum, this causes temporary denial of service for all reward distribution (StakingVault 68%, reinvestment 16%). The protocol cannot harvest until sufficient rewards accumulate.

### Recommendations

Handle small Buy & Burn amounts gracefully - either accumulate across cycles, redirect to reinvestment, or skip Buy & Burn for that cycle while continuing with other distributions.

## [L-07] `StakeETH` allows ETH to be staked while minting zero `wstETH`

### Description

`ValidatorVault.stakeETH(amount)` does not verify that wrapping the received stETH into wstETH produces a non-zero result.

Due to floor rounding in `wstETH.wrap()`, sufficiently small staking amounts may result in **zero wstETH minted**, even though ETH is successfully staked via Lido. This means the operation has a real external effect while producing no accounting-visible staking position.

This issue persists even if principal accounting is fully corrected.

### Recommendations

Ensure that `stakeETH` enforces a minimum staking amount or otherwise guarantees that staking operations result in a non-zero wstETH mint.

## [L-08] Unused token allowance after partial swap execution

### Description

`SwapActions` approves the full `amountIn` before swaps, but the retry mechanism reduces swap amounts on slippage errors, leaving unused allowance. For V4/PERMIT2 this persists until the deadline; for V3's `forceApprove` it remains indefinitely. Residual allowance could amplify other vulnerabilities.

### Recommendations

Approve only the final amount after retry calculations, or clean up post-swap.

## [L-09] Payout schedule advances only one cycle per call when cycles are overdue

### Description

`StakingVault._triggerCyclePayout` updates `nextCyclePayoutDay` incrementally by adding a single `cycleLength` ( `+= cycleLength` ). When a large number of days have passed without activity, this causes overdue payout schedules to “catch up” only one cycle per transaction.

As a result, even after a successful payout trigger, `nextCyclePayoutDay` can remain significantly behind `currentDay`. Processing all missed payout cycles requires multiple external calls to functions that invoke `_triggerPayouts()`, and reward distribution timing becomes dependent on subsequent user interactions after a period of inactivity.

This behavior does not lead to loss of funds or incorrect reward calculations, but it may result in delayed payouts and unintuitive operational behavior after long idle periods.

### Recommendations

Consider advancing `nextCyclePayoutDay` to the first cycle strictly after `currentDay` (e.g., via a loop or direct calculation) so that all overdue cycles are handled in a single call, avoiding reliance on repeated external interactions to catch up.

## [L-10] `CyclePayoutTriggered` logs `ethReward = 0` due to reward token ordering issue

### Description

In `StakingVault._triggerCyclePayout`, the `ethReward` value emitted in `CyclePayoutTriggered` is determined via a pre-scan over `rewardTokens` and therefore depends on token ordering. If the first token with a non-zero `tokenCyclePayoutPool` is an ERC20, the pre-scan sets `hasRewards = true` and breaks early, leaving `ethReward` as `0` even when the ETH pool for the same cycle is non-zero.

This can cause the `CyclePayoutTriggered(..., ethReward)` event to report `0` ETH while ETH rewards are actually distributed in the same call. This does not affect on-chain reward accounting, but can mislead off-chain indexers, dashboards, and analytics that rely on the `ethReward` event field.

**Pre-scan + early `break` makes `ethReward` order dependent**

```
uint256 ethReward = 0;
for (uint256 i = 0; i < tokensLength; i++) {
    address token = rewardTokens[i];
    if (tokenCyclePayoutPool[token][_cycleNo] > 0) {
        hasRewards = true;
        if (token == address(0)) {
            ethReward = tokenCyclePayoutPool[address(0)][_cycleNo];
        }
        break;
    }
}
```

**ETH distribution happens later regardless of the pre-scan**

```
if (tokenReward > 0) {
    cumulativeRewardPerShare[token][_cycleNo] += (tokenReward * PRECISION) /
_globalActiveShares;
```

```
    tokenCyclePayoutPool[token][_cycleNo] = 0;

    if (token == address(0)) {
        totalEthDistributed = totalEthDistributed + tokenReward;
    }
}
```

**Event uses the pre-scan `ethReward` value**

```
emit CyclePayoutTriggered(msg.sender, _cycleNo, ethReward);
```

### Recommendations

Make `ethReward` independent of `rewardTokens` ordering, e.g., compute and emit the actually distributed ETH amount from the main distribution loop (most accurate option), rather than relying on the pre-scan.

---

## [L-11] No `wstETH/stETH` oracle on ETH mainnet

### Description

`ValidatorVault` determines the current value of `wstETH` staked in the vault in terms of `ETH` by assuming that `stETH` trades 1:1 with ETH. While this assumption is already risky without any on-chain price deviation checks, the more fundamental vulnerability is the assumption that a valid `wstETH / stETH` price feed exists on the Ethereum mainnet.

If we carefully examine the oracle referenced by the protocol for `wstETH / stETH` in the test files, it is actually an AAVE oracle price adapter that returns the `wstETH / USD` price instead.

```
    function getWstETHRateFeedByChainId(uint256 chainId) internal pure returns (address) {
        //Ethereum - Chainlink wstEth/stETH rate feed
        if (chainId == 1) {
@>            return address(0x8B6851156023f4f5A66F68BEA80851c3D905Ac93);
        }
        //Sepolia - No official Chainlink feed, will use wstEth contract directly
        else if (chainId == 11155111) {
            return address(0); // Handled in deployment script
        } else if (chainId == 31337) {
            return address(0);
        } else {
            revert("Unsupported chainId");
        }
    }
```

Upon further review, it becomes clear that the Ethereum mainnet does not have any Chainlink `wstETH / stETH` price feeds. Such feeds only exist on L2 networks such as Base and Arbitrum.

This means that there is no price feed that correctly prices `wstETH` in terms of `stETH`. As a result, if a `rateFeed` is configured incorrectly, the calculated value of `wstETH` in terms of ETH, under the assumption that `stETH` is 1:1 with ETH, will always be incorrect.

In this scenario, the protocol effectively does not rely on a valid Chainlink price feed and instead depends entirely on a fallback mechanism that derives the price of `wstETH` in terms of `stETH`, which is then treated as 1:1 with ETH.

### Recommendations

The recommendation is to remove the code path that retrieves prices from Chainlink price feeds, since Ethereum mainnet does not have a `wstETH / stETH` oracle, and instead directly retrieve pricing data from the `stETH` contract. Additionally, a mechanism can possibly be introduced to cap the ratio returned by the `stETH` contract, similar to the CAPO framework used by AAVE.

## [L-12] Self-referral allows users to claim both minter and referrer bonuses

### Description

The `mint()` function lacks validation to prevent self-referral, allowing users to set themselves as the referrer and claim both the minter bonus and referral bonus, effectively doubling their bonus allocation and draining the referral pool faster than intended.

When a user mints tokens with a referrer, the protocol allocates two bonuses from the referral pool:

- Referral bonus (5% of minted tokens) to the referrer.
- Minter bonus (5% of minted tokens) to the minter.

The function accepts any address as the referrer without checking if it is the same as the minter ( `to` ) or the caller ( `msg.sender` ). This allows a user to:

1. Call `mint(to, to, vestingSchedule)` or `mint(to, msg.sender, vestingSchedule)` with themselves as the referrer.
2. Receive the normal tokens they are minting.
3. Receive the minter bonus (5%).
4. Receive the referral bonus (5%) by being their own referrer.

This effectively grants a 10% bonus instead of the intended 5% minter bonus, breaking the intent of the referral bonus to incentivize active users to refer new users. The issue manifests whenever a user mints tokens and sets themselves (either via `to` or `msg.sender`) as the referrer. Since the referral pool has a fixed allocation of 1 billion tokens, this abuse reduces the pool available for legitimate referrals and undermines the intended incentive structure.

### Recommendation

Add validation to prevent self-referral by requiring that the referrer is different from both the recipient and the caller:

if (referrer != ZERO_ADDRESS) { require(referrer != to, LaunchManager__SelfReferralNotAllowed()); require(referrer != msg.sender, LaunchManager__SelfReferralNotAllowed()); // ... rest of referral logic }

In addition, consider adding a mapping of "active users" who have already invested in the protocol and only limit the referral bonus to those users.

## [L-13] Farm removal fee distribution mismatch with documentation

### Description

We observe that during farm removals, all accumulated fees of `currency0` and `currency1` are distributed to stakers of 28-day cycles. However, the code comment states that ETH and WETH fees should be sent to the `taxRouter`, while other ERC20 token fees should be distributed to stakers during farm removals.

If one of the pool currencies is WETH or ETH, the current logic still distributes the accumulated fees to stakers instead of routing them to the `taxRouter`. This creates a mismatch between the documented behavior and the implemented logic.

If this behavior is intended, the recommendation is to update the code comment to reflect the actual implementation. If it is not intended, then for pools where one of the tokens is WETH or ETH, the accumulated fees should be sent to the `taxRouter` instead of being distributed to stakers.

```
    function removeFarm(bytes32 id, uint48 deadline)
        external
        beforeDeadline(deadline)
        validFarmId(id)
        nonReentrant
        onlyOwner
    {
        Farm storage farm = _farms.get(id);

        AccumulatedFees memory accumulatedFees;
        if (farm.lp.liquidity != 0) {
            (,, accumulatedFees) = _withdrawFromFarm(farm, farm.lp.liquidity, deadline);
        } else {
            accumulatedFees = _accumulatedFees[id];
        }
        uint256 amount0 = farm.currency0Balance;
        uint256 amount1 = farm.currency1Balance;

        address curr0 = Currency.unwrap(farm.poolKey.currency0);
        address curr1 = Currency.unwrap(farm.poolKey.currency1);
```

```
@>        // Handle withdrawal amounts: ETH and WETH go to taxRouter, other ERC20s go to
stakers
        uint256 ethAmount = farm.nativeBalance;
@>        uint256 stakerAmount0 = accumulatedFees.accumulatedFeeCurrency0;
@>        uint256 stakerAmount1 = accumulatedFees.accumulatedFeeCurrency1;
        // ...

        // Distribute all rewards to stakers at once (accumulated fees + ERC20 withdrawal
amounts)
@>        FarmKeeperRewardsLib.distributeRewards(stakingVault, stakerAmount0, stakerAmount1,
curr0, curr1);
```

## [L-14] Incorrect liquidity removed value in `FarmKeeper_Removed` event

### Description

When removing farms via `FarmKeeper::removeFarm()`, the `FarmKeeper_Removed` event emits an incorrect value for the `liquidityRemoved` field. Instead of indicating the actual liquidity that was removed, it emits `farm.lp.liquidity`, which has already been deleted at the time of emission. As a result, `liquidityRemoved` is always emitted as 0, which is incorrect.

The recommendation is to cache the initial liquidity value before deletion and emit that cached value as `liquidityRemoved` in the event.

```
        // ...
        delete _accumulatedFees[id];
        delete intermediaryPools[id];
        delete farmAllocPoints[id];

@>        _farms.remove(id);

        // Send ETH to tax router
        _safeETHTransfer(taxRouter, ethAmount);

@>        emit FarmKeeper_Removed(id, amount0, amount1, curr0, curr1, farm.lp.liquidity);
```

## [L-15] Constructor does not validate price configuration parameters

### Description

The `constructor()` directly assigns `_priceConfig` to `priceConfig` without performing any validation checks on the configuration parameters. In contrast, `setPriceConfig()` validates that the `weth` address is one of the tokens in the Uniswap pool by checking it against `token0` and `token1`. This inconsistency means an invalid configuration could be set during deployment, potentially causing failures when attempting to fetch prices from the

Uniswap fallback. The constructor should apply the same validation logic as `setPriceConfig()` to ensure the weth address corresponds to one of the pool tokens and that other configuration parameters are valid.

## [L-16] Intermediary pool configuration lacks validation

### Description

The `setIntermediaryConfiguration()` function allows the owner to update the intermediary pool configuration for a farm, but it does not validate that the provided `poolKey` corresponds to an actual existing pool. While `slippage` and `twapPeriod` are validated, the pool's existence is never checked. This differs from `enableV3Farm()` and `enableV4Farm()`, which validate pool existence during farm creation. Setting an invalid intermediary pool could cause reverts when attempting to swap native tokens through the non-existent pool via `swapNativeForFarm()` or `depositToFarm()`. Add validation to verify the pool exists based on the `isPoolV4` flag, similar to the validation performed in the farm enablement functions.

## [L-17] Twap period can be set to an invalid value

### Description

The `setTwapPeriod()` function in `OlympusXHook` allows the owner to set the TWAP period to any value, including zero or extremely low values, without validation. This period is used by `getTokenPriceInUsd()` and `getMarketCap()` to calculate time-weighted average prices from the Uniswap pool. Setting the TWAP period to zero or a very low value would break the TWAP oracle logic, making price calculations unreliable or causing reverts in the underlying TWAP library calls. Other contracts in the protocol, like `FarmKeeper`, validate the TWAP period to be between 1 minute and 60 minutes. Apply similar validation in `setTwapPeriod()` to ensure the TWAP period remains within a reasonable range that supports.

## [L-18] Reward tokens become permanently locked when removed with no active share

### Description

In `removeRewardToken()`, if outstanding rewards exist in cycle payout pools but `totalShares` is zero, the rewards cannot be distributed and become permanently locked in the contract. The distribution logic requires both conditions: `if (totalRemaining > 0 && totalShares > 0)`. When there are no active shares (all stakes have ended), the rewards in `tokenCyclePayoutPool` for all cycles cannot be distributed to stakers nor recovered by any mechanism. This edge case could occur if the protocol experiences a period with no active stakes while reward tokens are still being accumulated, or if a token needs to be removed

during such a period. Add a fallback mechanism to transfer any remaining rewards to a designated address (such as the genesis wallet or treasury) when `totalShares` is equal to zero.

## [L-19] Rounding dust from reward distribution becomes permanently locked

### Description

In `removeRewardToken()`, when distributing outstanding rewards across all cycles, the calculation `(amount * PRECISION) / totalShares` can result in rounding down, leaving dust tokens in the contract. After updating `cumulativeRewardPerShare`, the pool balance is set to zero without accounting for this rounding remainder. When users claim rewards, they receive `(cumulativeRewardPerShare * shares) / PRECISION`, which also rounds down. The combination of these two rounding operations means a small amount of tokens remains in the contract balance but is not reflected in claimable rewards. While typically negligible per distribution, these dust amounts accumulate over time and across all five cycles, becoming permanently unclaimable. Calculate the actual distributed amount and transfer any remaining dust to a designated address (genesis wallet or treasury) before zeroing the pool balances.

## [L-20] Summed reward claims across different tokens produce meaningless values

### Description

In `_claimAllTokensForCycle()`, the `totalClaimed` variable accumulates rewards across all reward tokens by summing `tokenReward` amounts. This value is returned and used in `claimAllRewards()` to populate the `claimedPerCycle` array. However, when multiple reward tokens exist with different decimal places (e.g., ETH with 18 decimals, USDC with 6 decimals), summing these amounts creates a meaningless value. For example, claiming 1 ETH (1e18) and 1000 USDC (1000e6) would result in `totalClaimed = 1e18 + 1000e6 = 1_000000001000000000`, which does not represent the actual value of either asset. The returned `claimedPerCycle` array from `claimAllRewards()` becomes unusable for off-chain tracking or display purposes when multiple tokens are registered. Consider tracking claimed amounts per token separately or removing the aggregated return value.

## [L-21] Inconsistent slippage handling for increase and decrease of liquidity in v4 l

### Description

In the v4 liquidity helper logic, two different slippage models are used for increase and decrease operations. During **decrease liquidity**, minimum output amounts ( `minAmount0` / `minAmount1` ) are adjusted using the configurable `farm.liquiditySlippage` parameter. In contrast, during **increase liquidity**, the calculated liquidity is always reduced by a fixed coefficient `9_950 / BPS` (0.5%), which is not configurable via farm parameters.

As a result, slippage tolerance is only partially parameterized. Operators may reasonably expect `farm.liquiditySlippage` to control slippage behavior consistently, but in practice it applies only to decrease operations, while increase operations always incur a hard-coded liquidity haircut.

This does not introduce a direct exploit, but creates operational and tuning risk: the fixed buffer may be suboptimal for certain pairs (volatility, decimals, liquidity depth) and can systematically reduce LP utilization, while its behavior is not adjustable through configuration.

**Decrease path uses configurable slippage**

```
minAmount0 = (minAmount0 * (BPS - farm.liquiditySlippage)) / BPS;
minAmount1 = (minAmount1 * (BPS - farm.liquiditySlippage)) / BPS;
```

**Increase path applies a fixed, non-configurable haircut**

```
liquidity = LiquidityAmounts.getLiquidityForAmounts(...);
liquidity = uint128(Math.mulDiv(liquidity, 9_950, BPS)); // fixed 0.5% buffer
```

### Recommendations

Make the increase-liquidity buffer configurable—either by reusing `farm.liquiditySlippage` where appropriate or by introducing a separate parameter (e.g., `increaseLiquidityBufferBps` ). Alternatively, explicitly document that increase operations always apply a fixed safety haircut while `farm.liquiditySlippage` applies only to decrease operations.

## [L-22] Implicit assumption on Chainlink price feed decimals in `EthPrice`

### Description

`EthPrice` implicitly assumes that the configured Chainlink price feed uses **18 decimals or fewer**. This assumption is not enforced or validated in code.

While this is true for standard ETH/USD Chainlink feeds today, the contract does not guard against misconfiguration. If a feed with **more than 18 decimals** were ever configured, price scaling logic would break or revert, potentially impacting all protocol paths that rely on ETH price (including minting and liquidity-related calculations).

This is not an exploitable vulnerability under correct configuration, but it represents a fragile external dependency assumption that could lead to systemic failures if violated.

### Recommendations

Explicitly validate that `decimals() <= 18` when interacting with the price feed (or at configuration time), or normalize prices defensively to a fixed internal precision with appropriate bounds checks.

## [L-23] `StakeETH` can be front run with a small amount to disrupt user transactions

### Description

The `stakeETH` function is callable by anyone and allows users to stake ETH held in ValidatorVault.sol into Lido. Although the function is permissionless and lets users specify the staking amount, the transaction will revert if the amount exceeds pendingEth:

```
        if (amount > pendingEth) revert ValidatorVault__InsufficientPendingEth();
```

Because of this behavior, a malicious user can front-run a transaction where stakeETH is called with an amount equal to pendingETH by submitting a transaction with a very small amount (e.g., 100 wei). This reduces pendingETH and causes the victim's transaction to revert.

While this issue does not lead to direct fund loss, it negatively impacts user experience and can degrade the protocol's reliability.

### Recommendations

When the specified amount is greater than pendingEth, cap the amount to `pendingEth` instead of reverting.

## [L-24] Missed slippage protection when users sell OLY

In Uniswap V4 pool manager, when users sell OLY tokens via `swap`, users will define the `sqrtPriceLimitX96` as the slippage protection. In OlympusXHook, when users sell OLY tokens, tax fees will be charged according to the current market cap.

For example, the current market cap is 26M; then the tax rate is 28%. As the seller, they will expect to lose around 28% of the output token. When the market cap becomes higher, the tax fee may become lower.

The problem here is that the market cap can be manipulated via burning OLY from the BuyAndBurn contract. It is possible to cause the tax fee rate to become larger by manipulation. This will cause traders to pay more tax than expected.

```
    function _beforeSwap(address sender, PoolKey calldata key, SwapParams calldata params,
bytes calldata)
        internal
        override
        returns (bytes4, BeforeSwapDelta, uint24)
    {
        uint256 netRequested = uint256(int256(params.amountSpecified));
        uint256 taxBps = getCurrentMarketSellTaxBps();
        uint256 taxAmount = (netRequested * taxBps) / BPS;
    }
```

### Recommendations

Add one tax protection when users swap via the hook.

## [L-25] Mint accounting break with mixed liquidity and vesting configuration

During the minting of `OLY` tokens via `LaunchManager::mint()`, if a vesting schedule is configured with `0 < liquidTokenBps < 100` and the `tiers.percentageBps` values sum up to 100 percent, the accounting logic breaks.

The core issue is that the accounting always uses `tokensToMint` to calculate the amount of `OLY` tokens to be staked on behalf of the user in the `StakingVault`, instead of using the remaining amount after liquid tokens are accounted for.

Consider the following scenario.

1. `schedule.liquidTokenBps = 6000`.
2. `tokensToMint = 100e18`.
3. `vestingSchedules[0].tiers[0] = ILaunchManager.VestingTier({percentageBps: 2500, stakingDays: 369})`.
4. `vestingSchedules[0].tiers[1] = ILaunchManager.VestingTier({percentageBps: 5000, stakingDays: 184})`.
5. `vestingSchedules[0].tiers[2] = ILaunchManager.VestingTier({percentageBps: 2500, stakingDays: 88})`.

The calculations then proceed as follows.

1. `liquidTokens = 100e18 * 6000 / 10000 = 60e18`.
2. `amounts[0] = 100e18 * 2500 / 10000 = 25e18`.
3. `amounts[1] = 100e18 * 5000 / 10000 = 50e18`.
4. `amounts[2] = 100e18 - 60e18 - 25e18 - 50e18` which underflows.

This underflow prevents any `OLY` tokens from being minted even when the user supplies ETH.

```
    function mint(address to, address referrer, uint256 vestingSchedule) external payable
override nonReentrant {
        // ...

        uint256 liquidTokens;
        if (schedule.liquidTokenBps > 0) {
@>            liquidTokens = (tokensToMint * schedule.liquidTokenBps) / BPS;
        }
        IERC20(address(olympusX)).transfer(to, tokensToMint + minterBonus);

        uint256 tiersLength = schedule.tiers.length;
        uint256[] memory amounts = new uint256[](tiersLength);
        uint16[] memory stakeDays = new uint16[](tiersLength);
        uint256 runningSum;
        for (uint256 i; i < tiersLength; i++) {
            if (i == tiersLength - 1) {
@>                amounts[i] = tokensToMint - liquidTokens - runningSum;
            } else {
                amounts[i] = (tokensToMint * schedule.tiers[i].percentageBps) / BPS;
                runningSum = runningSum + amounts[i];
            }
            stakeDays[i] = schedule.tiers[i].stakingDays;
        }
        // ...
```

### Recommendations

The recommendation is to correct the accounting by using the remaining amount, namely `tokensToMint` minus the liquid token portion defined by `liquidTokenBps`, when calculating the amounts to be staked in the `StakingVault`.

## [L-26] Insufficient observation cardinality validation in `FarmKeeper.setTwapPeriod`

### Description

The `FarmKeeper.setTwapPeriod()` function allows the owner to update the TWAP period for a farm without validating whether the pool has sufficient observation cardinality to support the requested period. This creates a vulnerability where TWAP price queries may fail and price determination will fall back to manipulable spot prices.

When `setTwapPeriod()` increases the TWAP period (e.g., from 15 minutes to 60 minutes), it does not verify that the pool's `observationCardinality` can accommodate the longer lookback window. The required number of observations depends on the block time and TWAP period. For example, on Ethereum mainnet with a 12-second block time, a 60-minute TWAP

requires approximately 300 observations (3600 seconds / 12 seconds). However, pools are typically initialized with only 150 observations on mainnet (as seen in `V4OracleLib.getInitialCardinality()`).

When the observation cardinality is insufficient, key functionality of the FarmKeeper will fail:

**For both V3 and V4 Pools**: The call to `V3OracleLib.getTwapSqrtPriceX96()` or `V4OracleLib.getTwapSqrtPriceX96()` will revert with `TooShortPoolCardinality()` when `getOldestObservationSecondsAgo()` returns a value less than the requested `twapPeriod`. This causes all liquidity operations to fail, including `decreaseLiquidity()` and liquidity additions (via `addLiquidity()`), effectively locking funds in the farm.

### Recommendation

Add observation cardinality validation to `setTwapPeriod()` that checks whether the pool can support the requested TWAP period and automatically increases the cardinality if needed. Additionally, consider adding the same validation to farm creation in `addFarm()` and `addLaunchFarm()` to ensure pools have sufficient cardinality from the start.

## [L-27] ValidatorVault assumes `stETH` is pegged 1 to 1 with `ETH`

### Description

We observe that in multiple instances throughout `ValidatorVault`, the protocol assumes that `stETH` is pegged 1:1 with ETH. While both assets tend to trade close to parity, this has not always been the case. Market conditions can cause `stETH` to deviate from ETH, breaking the assumed peg.

We observe the following instances where `stETH` is treated as 1:1 with ETH.

1. In `ValidatorVault::harvest()`, when calculating `currentEthValue` of the total `wstETH` held by the `ValidatorVault` in terms of ETH, `getWstETHValue()` assumes `stETH` is 1:1 with ETH.
2. In `ValidatorVault::harvest()`, when calculating `reinvestedETHValue` for the portion of `wstETHToReinvest`, the `wstETH` amount is converted to a `stETH` value and `stETH` is again treated as 1:1 with ETH.
3. In `ValidatorVault::_getFallbackValue()`, `stETH` is treated as 1:1 with ETH.

One impact of this assumption is that if `stETH` is worth less than ETH, the value of `wstETH` will be overstated. This can negatively affect reward accounting during harvesting. An overstated ETH value for `wstETH` will cause `rewardsInETH` to be overstated, resulting in more `wstETH` being burned and distributed than intended, which is problematic.

```
    function harvest() external nonReentrant {
        if (block.timestamp < lastHarvestTime + HARVEST_CYCLE) revert
ValidatorVault__HarvestTooEarly();
```

```
        // Get current wstEth contract
        IWstETH wstETHContract = IWstETH(wstEth);

        // Calculate current ETH value of wstEth holdings
@>        uint256 currentEthValue = getWstEthValue();

        // Calculate rewards in ETH terms (value appreciation)
        if (currentEthValue <= totalWstETHPrincipal) {
            // No rewards, just update timestamp and emit event
            lastHarvestTime = uint48(block.timestamp);
            emit RewardsHarvested(currentHarvestCycle, 0, 0, 0, 0, totalWstETHPrincipal);
            unchecked {
                currentHarvestCycle++;
            }
            return;
        }

        uint256 rewardsInETH;
        unchecked {
            rewardsInETH = currentEthValue - totalWstETHPrincipal;
        }

        // Convert rewards to wstEth amount
@>        uint256 rewardWstETH = wstETHContract.getWstETHByStETH(rewardsInETH);

        // ...

        // Calculate reinvested value in ETH terms
@>        uint256 reinvestedETHValue = wstETHContract.getStETHByWstETH(wstETHToReinvest);
        unchecked {
@>            totalWstETHPrincipal = totalWstETHPrincipal + reinvestedETHValue; // Add
reinvested ETH value to principal

            // Track total rewards harvested
            totalRewardsHarvested = totalRewardsHarvested + rewardsInETH;
        }

        // Update cycle and timestamp
        lastHarvestTime = uint48(block.timestamp);

        emit RewardsHarvested(
            currentHarvestCycle,
            rewardsInETH,
            wstETHToStakers,
            wstETHToBuyAndBurn,
            wstETHToReinvest,
            totalWstETHPrincipal
        );

        unchecked {
            currentHarvestCycle++;
        }
    }
```

```solidity
    function _getFallbackValue() internal view returns (uint256) {
        // Use wstEth contract method to get stETH value
```

```
        IWstETH wstEthContract = IWstETH(wstEth);
@>        uint256 stEthValue = wstEthContract.getStETHByWstETH(totalWstETHStaked);
        // stETH is 1:1 with ETH
@>        return stEthValue;
    }
```

```solidity
    function getWstEthValue() public view returns (uint256) {
        if (totalWstETHStaked == 0) return 0;

        if (address(rateFeed) == address(0)) {
            return _getFallbackValue();
        }

        // Get decimals from rate feed (with fallback to 18 if unavailable)
        uint8 feedDecimals = 18; // Default to 18 decimals
        try rateFeed.decimals() returns (uint8 decimals) {
            feedDecimals = decimals;
        } catch {
            // If decimals() fails, use default 18
        }

        // Try to get rate from Chainlink feed first
        try rateFeed.latestRoundData() returns (
            uint80, /* roundId */
            int256 answer,
            uint256, /* startedAt */
            uint256 updatedAt,
            uint80 /* answeredInRound */
        ) {
            // Check for valid answer
            // Check staleness
            if (answer <= 0 || block.timestamp - updatedAt > stalenessThreshold) {
@>                return _getFallbackValue();
            }

            // Chainlink feed gives wstEth/stETH rate
            // We need to convert our wstEth to stETH value
            // answer is scaled by feedDecimals, so normalize to 18 decimals for calculation
            uint256 rate = uint256(answer) * (10 ** (18 - feedDecimals));
            uint256 stEthValue = Math.mulDiv(totalWstETHStaked, rate, 1e18);

            // stETH trades 1:1 with ETH (per Lido documentation)
@>            return stEthValue;
        } catch {
@>            return _getFallbackValue();
        }
    }
```
## Recommendations

The recommendation is to obtain the `stETH` to ETH price from Chainlink price feeds.