Pashov Audit Group

DomFi Security Review

December 7th 2025 - January 3rd 2026

Contents

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. About DomFi

DomFi is a leveraged perpetual trading protocol where traders open long or short positions using collateral, while an ERC-4626 vault acts as the counterparty by pooling liquidity and absorbing realized profit and loss. DomFi is implemented as a modular smart-contract system handling position state, oracle-driven price execution, funding and rollover fees, epoch-based PnL settlement, and vault share repricing with optional NFT-based locked deposits.

5. Executive Summary

A time-boxed security review of the **domination-finance/domfi-contracts** repository was done by Pashov Audit Group, during which **Tejas Warambhe**, **0xunforgiven**, **IvanFitro**, **0xAlix2**, **ExtraCaterpillar** engaged to review **DomFi**. A total of **67** issues were uncovered.

Protocol Summary

Project Name	DomFi
Protocol Type	Perpetual Trading
Timeline	December 7th 2025 - January 3rd 2026

Review commit hash:
- 063fc31a361ef0461085e3ca8ecf2f65acaec1b1 (domination-finance/domfi-contracts)

Fixes review commit hash:
- 53a3dc92a94786705bbfba8c0d0b0996adf3c0de (domination-finance/domfi-contracts)

Full timeline:

December 7th - January 3rd - main audit phase
January 3rd - January 18th - fixes review phase
January 19th - January 27th - single researcher extra audit and fixes review

Scope

`Delegatable.sol` `interfaces/` `ChainUtils.sol` `TradingCallbacksLib.sol` `DomfiLockedDepositNft.sol` `DomfiOpenPnl.sol` `DomfiOracle.sol` `DomfiPairInfos.sol` `DomfiPairsStorage.sol` `DomfiPriceRouter.sol` `DomfiPriceUpKeep.sol` `DomfiPrivatePriceUpKeep.sol` `DomfiRegistry.sol` `DomfiTimelockManager.sol` `DomfiTimelockOwner.sol` `DomfiTradesUpKeep.sol` `DomfiTrading.sol` `DomfiTradingCallbacks.sol` `DomfiTradingStorage.sol` `DomfiVault.sol` `DomfiVerifier.sol`

6. Findings

Findings count

Severity	Amount
High	9
Medium	29
Low	29
Total findings	**67**

Summary of findings

ID	Title	Severity	Status
[H-01]	Trading storage is not designed to handle native token transfers	High	**Resolved**
[H-02]	Closing fee direction is incorrect	High	**Resolved**
[H-03]	Wrong `priceAfterImpact` calculation in `getTradePriceImpact`	High	**Resolved**
[H-04]	Price impact uses improper trade value	High	**Resolved**
[H-05]	UnlockDeposit() adds discounted assets as traders profit and loss	High	**Resolved**
[H-06]	Code does not scale `accPnlPerToken` in `scaleVariables` always	High	**Resolved**
[H-07]	`unlockDeposit()` shouldn't change `totalAccPnl`	High	**Resolved**
[H-08]	`updateAccPnlPerTokenUsed()` code should multiply `delta` by 1e6	High	**Resolved**
[H-09]	`TradeInfo` struct setup is wrong in `registerTrade()` function	High	**Resolved**
[M-01]	Avoid oracle fees by setting TP/SL close to market price	Medium	**Resolved**
[M-02]	Handle `handleRemoveCollateral()` fails to unregister trigger causing DoS	Medium	**Resolved**
[M-03]	Closing fee is calculated after updating the open interest	Medium	**Resolved**

ID	Title	Severity	Status
[M-04]	Donation attack through `distributeReward` to inflate share price	Medium	Resolved
[M-05]	No slippage control for withdraw and redeem in the vault	Medium	Resolved
[M-06]	Function `removePendingAutomationOrder` would revert	Medium	Resolved
[M-07]	`executeAutomationOrder()` lacks order time checks and enables frontrunning	Medium	Resolved
[M-08]	Minimum position size can be bypassed in `openTrade` function	Medium	Resolved
[M-09]	Execute `executeAutomationOrder()` ignores SL/TP last update timestamp	Medium	Resolved
[M-10]	Trade validation performed with pre-fee collateral	Medium	Resolved
[M-11]	Inconsistent price usage between OI validation and storage	Medium	Resolved
[M-12]	Automation close order validation uses an inconsistent price	Medium	Resolved
[M-13]	Timed-out collateral orders not cleaned when trigger pending	Medium	Resolved
[M-14]	No slippage protection on market close orders	Medium	Resolved
[M-15]	Limit order execution does not validate SL TP placement	Medium	Resolved
[M-16]	Price impact miscalculation on partial market close	Medium	Resolved
[M-17]	Pending automation order for OPEN limit can be removed before timeout	Medium	Resolved
[M-18]	Uncapped closing fees can cause underflow and block trade closure	Medium	Resolved
[M-19]	`updateAccPnlPerTokenUsed()` only applies positive unrealized PnL	Medium	Resolved
[M-20]	Code does not reset `orderTriggerBlock` when updating trades	Medium	Resolved
[M-21]	Using average for unrealized PnL settlement gives opportunity to extract value	Medium	Resolved

ID	Title	Severity	Status
[M-22]	Discrepancy for `isLiquidated` in trade and automation callbacks	Medium	Acknowledged
[M-23]	Attacker can front-run unrealized PnL as it takes effect at the end of epoch	Medium	**Resolved**
[M-24]	Front-running `unlockDeposit` can avoid loss distribution	Medium	**Resolved**
[M-25]	Inconsistent price impact calculation	Medium	**Resolved**
[M-26]	DepositWithDiscountAndLock and mintWithDiscountAndLock missing slippage c	Medium	**Resolved**
[M-27]	Using max for share price calculation is wrong	Medium	**Resolved**
[M-28]	Variables `tpLastUpdated` and `slLastUpdated` should be validator	Medium	**Resolved**
[M-29]	`unregisterTrade()` shouldn't remove `LimitOrder.OPEN`	Medium	**Resolved**
[L-01]	Redundant validation in `updateOpenLimitOrder`	Low	**Resolved**
[L-02]	Position mutations are allowed even under pending market closure	Low	**Resolved**
[L-03]	DomfiTradingStorage handleOracleFee() returns cumulative oracleFees	Low	**Resolved**
[L-04]	SetDev() fails to collect fees for the old `dev` address	Low	**Resolved**
[L-05]	ReceiveAssets() missing onlyCallbacks allows public vault donation	Low	**Resolved**
[L-06]	Override `_transferOwnership()` in `DomfiRegistry` to avoid role overlap	Low	**Resolved**
[L-07]	setValue() and setValues() use `<` instead of `<=` to validate the timestamp	Low	**Resolved**
[L-08]	FeeOk does not check whether `name != bytes(0)`	Low	**Resolved**
[L-09]	ScaleVariables() division by zero prevents full withdrawal	Low	**Resolved**

ID	Title	Severity	Status
[L-10]	Remove `removeCollateral` and `topUpCollateral` may reduce position below limit	Low	**Resolved**
[L-11]	OpenTradeMarketCallback() uses prefee collateral for price impact calculation	Low	**Resolved**
[L-12]	Tp/sl validation ignores pending remove-collateral operation	Low	**Resolved**
[L-13]	Callbacks can execute on stale state after trigger timeout	Low	Acknowledged
[L-14]	Open trade does not validate collateral sufficiency for opening fee	Low	Acknowledged
[L-15]	Pause check in `executeAutomationOrder` reverts instead of returning status	Low	**Resolved**
[L-16]	Redundant `cancelReason` condition in `closeTradeMarketCallback`	Low	**Resolved**
[L-17]	Remove `removeCollateral` does not validate `newLeverage`	Low	**Resolved**
[L-18]	DomfiVault::maxRedeem can underflow violating ERC4626 expectations	Low	**Resolved**
[L-19]	Inconsistent rounding of `accPnlPerToken` in `updateAccPnlPerTokenUsed`	Low	**Resolved**
[L-20]	`PairInfos::getTradeLiquidationPrice` does not return correct liquidation price	Low	**Resolved**
[L-21]	Function `removePendingRemoveCollateral` would always revert	Low	**Resolved**
[L-22]	Oracle fee cap enables griefing via dust orders draining subsidized gas	Low	**Resolved**
[L-23]	Deposit/withdraw should not change share prices	Low	**Resolved**
[L-24]	Deposit and mint functions may validate against outdated max supply	Low	**Resolved**
[L-25]	Allowance bypass in `makeWithdrawRequest` and `cancelWithdrawRequest`	Low	**Resolved**
[L-26]	Remove `removePair()` can be DOSed by a malicious user	Low	Acknowledged

ID	Title	Severity	Status
[L-27]	`unlockDeposit()` should check `getPnlPerToken() > maxAccPnlPerToken()`	Low	Resolved
[L-28]	"Check if SL/TP is reached" is repeated two times	Low	Resolved
[L-29]	`startNewEpoch()` blends `spotPnl` even when median fallback is triggered	Low	Resolved

High findings

[H-01] Trading storage is not designed to handle native token transfers

Severity

Impact: Medium

Likelihood: High

Description

The `DomfiTradingStorage::handleOracleFee()` function is used in `DomfiTrading::openTrade()`, `DomfiTrading::closeTradeMarket()`, and `DomfiTrading::removeCollateral()` to collect the oracle fees from the user and send the same to the trading storage contract:

```
function handleOracleFee(uint256 amount)
    external
    onlyTrading
    returns (uint256 updatedOracleFees)
{
    oracleFees += amount;
    updatedOracleFees = oracleFees;
}
```

However, neither the trading storage contract is capable of handling the native token, nor is the function marked payable in nature. This will lead to reverts in all three functions mentioned above.

Recommendations

It is recommended to mark the `handleOracleFee()` function as payable in nature and update all the calls in the `DomfiTrading` contract:

```
function handleOracleFee(uint256 amount)
    external
+        payable
    onlyTrading
    returns (uint256 updatedOracleFees)
{
    oracleFees += amount;
    updatedOracleFees = oracleFees;
}
```

```
-        uint256 oracleFeePaid = tradingStorage.handleOracleFee(oracleFeeOffered);
+        uint256 oracleFeePaid = tradingStorage.handleOracleFee{ value: oracleFeeOffered }
(oracleFeeOffered);
```

[H-02] Closing fee direction is incorrect

Severity

Impact: Medium

Likelihood: High

Description

The `DomfiTradingStorage::handleClosingFees()` function is used to calculate the closing fees as per the position size and open interest delta:

```
    function handleClosingFees(
        uint16 pairIndex,
        uint256 latestPrice,
        uint256 leveragedPositionSize,
        uint32 leverage,
        bool isBuy
    ) external onlyCallbacks returns (uint256 devFee, uint256 vaultFee) {
        uint256 oiLong = (openInterest[pairIndex][uint256(OpenInterest.LONG)] * latestPrice)
            / PRECISION_18 / 1e12;
        uint256 oiShort = (openInterest[pairIndex][uint256(OpenInterest.SHORT)] * latestPrice)
            / PRECISION_18 / 1e12;

        int256 oiDelta = oiLong.toInt256() - oiShort.toInt256();

        (devFee, vaultFee) =
IDomfiPairInfos(registry.getContractAddress("pairInfos")).getClosingFee(
            pairIndex,
            isBuy ? leveragedPositionSize.toInt256() : -leveragedPositionSize.toInt256(),
<<@
            leverage,
            oiDelta
        );

        devFees += devFee;
    }
```

We can consider applying a fee to be equivalent to taking a trade in the opposite direction of the closing trade. However, as we can observe that the closing fee being applied uses a positive `leveragedPositionSize.toInt256()` amount in the case of a buy trade being closed, which is contradictory to the closing fee being applied.

In the current logic, if `makerFeeP` < `takerFeeP`, traders who rebalance skew on close are overcharged, and traders whose closes worsen skew are undercharged.

Recommendations

It is recommended to fix the direction of fees being applied:

```
    function handleClosingFees(
        uint16 pairIndex,
        uint256 latestPrice,
        uint256 leveragedPositionSize,
        uint32 leverage,
        bool isBuy
    ) external onlyCallbacks returns (uint256 devFee, uint256 vaultFee) {
        uint256 oiLong = (openInterest[pairIndex][uint256(OpenInterest.LONG)] * latestPrice)
            / PRECISION_18 / 1e12;
        uint256 oiShort = (openInterest[pairIndex][uint256(OpenInterest.SHORT)] * latestPrice)
            / PRECISION_18 / 1e12;

        int256 oiDelta = oiLong.toInt256() - oiShort.toInt256();

        (devFee, vaultFee) =
IDomfiPairInfos(registry.getContractAddress("pairInfos")).getClosingFee(
            pairIndex,

-             isBuy ? leveragedPositionSize.toInt256() : -leveragedPositionSize.toInt256(),
+             isBuy ? -leveragedPositionSize.toInt256() : leveragedPositionSize.toInt256(),
            leverage,
            oiDelta
        );

        devFees += devFee;
    }
```

[H-03] Wrong `priceAfterImpact` calculation in `getTradePriceImpact`

Severity

Impact: Medium

Likelihood: High

Description

In the function `getTradePriceImpact()`, after calculating the spread, the code calculates `priceAfterImpact` based on the `isLong` value:

```
        // round up to avoid truncation
        uint256 adjustment = priceU256.mulDiv(effectiveSpread, PRECISION_18,
Math.Rounding.Ceil);

        // potential underflows are handled natively
        priceAfterImpact = isLong ? priceU256 + adjustment : priceU256 - adjustment;
```

```
    // multiply absolute value by `PRECISION_18` and `100`
    priceImpactP = SignedMath.abs(price - int256(priceAfterImpact)) * 1e20 / priceU256;
```

Code assumes that if trade is `Long` / `Short` then it should always increase/decrease the price. The issue is that this function is being called for closing positions too, and while the position is `Long` it can be closed, and price impact should decrease the price (as closing a Long position is like selling). As a result of this issue, code always increases the opening price for `Long` positions, and it will give the attacker the ability to extract value: 1- While the net OI is negative, the attacker will open a `Long` position with volume equal to OI imbalance, and because net OI would be zero, the price impact would be 0 too. 2- Then the attacker would close their position, and this time the net OI would be bigger than zero, and price impact would be bigger than zero, but because the code always increases the price for `Long` positions, the resulting `priceAfterImpact` would be higher than the attacker's position's open price, and as a result, the attacker would close the position with a profit.

Recommendations

To determine the `priceAfterImpact`, check both `isLong` and `isOpen` and determine the price movement direction based on both of those parameters.

[H-04] Price impact uses improper trade value

Severity

Impact: Medium

Likelihood: High

Description

The `DomfiTrading::closeTradeMarket()` stores the user's pending market close orders:

```
function closeTradeMarket(uint16 pairIndex, uint8 index, uint16 closePercentage)
    external
    payable
    notDone
{
   // . . .
    tradingStorage.storePendingMarketOrder(
        IDomfiTradingStorage.PendingMarketOrder(
            0,
            0,
            0,
            IDomfiTradingStorage.Trade(0, 0, 0, 0, sender, 0, pairIndex, index, t.buy),
            closePercentage
        ),
        orderId,
        false
    );
```

```
        emit MarketCloseOrderInitiated(orderId, i.tradeId, sender, pairIndex, closePercentage);
    }
```

The `Trade` struct here stores values such as `collateral`, `openPrice`, `tp`, `sl`, and `leverage` as 0.

This market order is supposed to be fulfilled via `DomfiTradingCallbacks::closeTradeMarketCallback()`:

```
    function closeTradeMarketCallback(IDomfiPriceUpKeep.PriceUpKeepAnswer calldata a)
        external
        notDone
    {
       // . . .
        IDomfiTradingStorage.Trade memory t =
            tradingStorage.getOpenTrade(trade.trader, trade.pairIndex, trade.index);

        CancelReason cancelReason = t.leverage == 0 ? CancelReason.NO_TRADE : CancelReason.NONE;

        IDomfiTradingStorage.TradeInfo memory i =
            tradingStorage.getOpenTradeInfo(t.trader, t.pairIndex, t.index);

        if (cancelReason != CancelReason.NO_TRADE && cancelReason == CancelReason.NONE) {
            (uint256 priceImpactP, uint256 priceAfterImpact) = TradingCallbacksLib
                .getTradePriceImpact(                              <<@
                a.price,
                trade.collateral,
                trade.leverage,
                tradingStorage,
                trade.pairIndex,
                false,
                trade.buy
            );

         // . . .
    }
```

However, the `getTradePriceImpact()` function utilizes the trade struct, which is expected to be empty. As a result, zero values for collateral and leverage are passed, leading to an incorrect price impact calculation.

Recommendations

It is recommended to pass the open trade's structure instead:

```
            (uint256 priceImpactP, uint256 priceAfterImpact) = TradingCallbacksLib
                .getTradePriceImpact(
                a.price,

-                trade.collateral,
+               t.collateral,

-                trade.leverage,
+                t.leverage,
                tradingStorage,

-                trade.pairIndex,
+                t.pairIndex,
                false,

-                trade.buy
+                t.buy
            );
```

[H-05] UnlockDeposit() adds discounted assets as traders profit and loss

Severity

Impact: Medium

Likelihood: High

Description

Users can deposit their funds with a lock and receive extra shares. Later when those locks expire, code adds the discounted assets to the variable `accPnlPerToken` and updates the share price to socialize the discounted assets.

```
        int256 accPnlDelta = d.assetsDiscount.mulDiv(PRECISION_18, totalSupply(),
Math.Rounding.Ceil).toInt256();

        accPnlPerToken += accPnlDelta;
        if (accPnlPerToken > maxAccPnlPerToken().toInt256()) {
            revert NotEnoughAssets();
        }

        lockedDepositNft.burn(depositId);

        accPnlPerTokenUsed += accPnlDelta;
        updateShareToAssetsPrice();
```

This creates multiple issues:

1. Variable `accPnlPerToken` is used to track traders' profit and loss, and the code uses net profit and loss to calculate share price if only the accumulated value is positive. Adding discounted assets to `accPnlPerToken` interferes with this accumulated net profit and loss of traders, which works as a liquidity buffer.
2. Changing variable `accPnlPerToken` does not change share price when `accPnlPerToken < 0`. It means adding discounted deposits to `accPnlPerToken` will not change share price in those cases, and as a result, the vault would become insolvent because total supply includes the discounted assets, but share price does not reflect that, and `supply * price` would be higher than total assets.

Recommendations

Distribute discounted assets by decreasing the variable `accRewardsPerToken`.

[H-06] Code does not scale `accPnlPerToken` in `scaleVariables` always

Severity

Impact: Medium

Likelihood: High

Description

In function `scaleVariables()`, code scales the value of `accPnlPerToken` to reflect the changed balance:

```
        uint256 supply = totalSupply();

        if (accPnlPerToken < 0) {
            accPnlPerToken = accPnlPerToken * supply.toInt256()
                / (isDeposit ? (supply + shares).toInt256() : (supply - shares).toInt256());
        }
    }
```

The issue is that when `accPnlPerToken > 0`, code does not update the value, and as a result the implied accumulated PnL would be wrong. For example:

1. Suppose there are 100 tokens and 100 shares in the vault, and traders' profit is 10 tokens, so `accPnlPerToken` would be 0.1, and share price would be 0.9.
2. User A deposits 90 tokens and would receive 100 shares, and the code will not update `accPnlPerToken` because its value is positive.
3. Now total shares would be 200, and `accPnlPerToken` would be 0.1, and the implied accumulated PnL would be `200 * 0.1 = 20`, which is wrong.

The impact is that code would use the wrong value for PnL, and the share price and yield calculations would be wrong over time.

Recommendations

Always scale the value of the `accPnlPerToken` in the `scaleVariables()` function.

[H-07] `unlockDeposit()` shouldn't change `totalAccPnl`

In the new design, the discount is subtracted from `accRewardsPerToken` and there's no need to change value of the `totalAccPnl` at all. Remove the line that increase `totalAccPnl` to fix the issue.

```
        totalAccPnl += discount.toInt256();

        if (getPnlPerToken() > maxAccPnlPerToken().toInt256()) {
            revert NotEnoughAssets();
        }

        accRewardsPerToken -= uint256(discount);
```

[H-08] `updateAccPnlPerTokenUsed()` code should multiply `delta` by 1e6

The variables `tradenotional` and `openPnl` have 12 decimal and `totalAccPnl` have 18 decimal so when calculating delta `openPnl` and adding it to `totalAccPnl` code should multiply the delta value by 1e6. To fix the issue multiply `delta` by 1e6 when adding it to `totalAccPnl` in the line `totalAccPnl += delta;`.

```
        int256 delta = newOpenPnl - prevOpenPnl;
        uint256 supply = totalSupply();

        int256 maxAccPnl = (
            Math.min(
                uint256(maxAccPnlPerToken().toInt256() - getPnlPerToken()) * supply /
PRECISION_6,
                maxAccOpenPnlDeltaPerToken * supply / PRECISION_6
            )
        ).toInt256();

        delta = delta > maxAccPnl ? maxAccPnl : delta;

        totalAccPnl += delta;
```

[H-09] `TradeInfo` struct setup is wrong in `registerTrade()` function

The struct `TradeInfo` fields changed and the `initialLeverage` is stored after the `tpLastUpdated` and `slLastUpdated` values:

```
    struct TradeInfo {
        uint256 tradeId;
        uint256 oiNotional; // PRECISION_18
        uint256 tpLastUpdated; // Should be `validator`
        uint256 slLastUpdated; // Should be `validator`
        uint32 initialLeverage;
        uint32 createdAt;
        bool beingMarketClosed;
    }
```

The issue is that code use incorrect orders when creating a `TradeInfo` object and. The leverage is before the timestamp variables:

```
        tradingStorage.storeTrade(
            trade,
            IDomfiTradingStorage.TradeInfo(
                tradeId,
                trade.collateral * uint256(1e12) * trade.leverage / PRECISION_2 * PRECISION_18
                    / trade.openPrice,
                trade.leverage,
                currBlock,
                currBlock,
                currBlock,
                false
            )
        );
```

To fix this, either update the `TradeInfo` construct or update the function `registerTrade()` and make the variable orders the same.

Medium findings

[M-01] Avoid oracle fees by setting TP/SL close to market price

Severity

Impact: Low

Likelihood: High

Description

In general, users can close their trades via `DomfiTrading::closeTradeMarket()`, which would close their running positions at the market price instantly. However, closing positions via `closeTradeMarket()` would require paying an oracle fee, which can be gamed by certain actors by using the `DomfiTrading::updateTp()` and `DomfiTrading::updateSl()`.

A user can observe the current market price of the market and apply the TP and SL at that exact mark or simply squash it just above and below that price to ensure that the automation trigger invokes immediately, giving them a market closing for the order.

Recommendations

It is recommended to charge an oracle fee for updating the TP and SL via `updateTp()` / `updateSl()`.

[M-02] Handle `handleRemoveCollateral()` fails to unregister trigger causing DoS

Severity

Impact: Low

Likelihood: High

Description

The `DomfiTradingCallbacks::handleRemoveCollateral()` is used by the keeper to fulfill the remove collateral request made by the user. This function first tries to check for any sort of discrepancy that could lead to the trade's cancellation; otherwise, it updates the trade and removes the pending trigger:

```
function handleRemoveCollateral(IDomfiPriceUpKeep.PriceUpKeepAnswer calldata a)
        external
        notDone
```

```
    {
       // . . .
        tradingStorage.transferUsdc(address(tradingStorage), request.trader,
request.removeAmount);
        tradingStorage.updateTrade(trade);
        pairsStorage.updateGroupCollateral(trade.pairIndex, request.removeAmount, trade.buy,
false);

        tradingStorage.unregisterPendingRemoveCollateral(a.orderId);      <<@

        tradingStorage.unregisterTrigger(            <<@
            request.trader,
            request.pairIndex,
            request.index,
            IDomfiTradingStorage.LimitOrder.REMOVE_COLLATERAL
        );
      // . . .
```

However, in a case where the `cancelReason != CancelReason.NONE`, the function calls `unregisterPendingRemoveCollateral()` to delete the `orderId` from the `pendingRemoveCollaterals` mapping, but fails to call `unregisterTrigger()`.

```
        if (cancelReason != CancelReason.NONE) {
            emit RemoveCollateralRejected(
                a.orderId,
                tradeInfo.tradeId,
                request.trader,
                request.pairIndex,
                request.removeAmount,
                cancelReason
            );

            tradingStorage.unregisterPendingRemoveCollateral(a.orderId);     <<@

            return;
        }
```

In such a situation, no other clean-up path can be used to remove this particular trigger. If we try to remove the pending trigger via `DomfiTrading::removePendingRemoveCollateral()`, it will not work as the pending entry is missing:

```
    function removePendingRemoveCollateral(uint256 orderId)
        external
        onlyTradesUpKeep
        notDone
        returns (IDomfiTrading.AutomationOrderStatus)
    {
        IDomfiTradingStorage tradingStorage =
            IDomfiTradingStorage(registry.getContractAddress("tradingStorage"));

        IDomfiTradingStorage.PendingRemoveCollateral memory pending =
            tradingStorage.getPendingRemoveCollateral(orderId);

        if (pending.trader == address(0)) {          <<@
```

```
            return IDomfiTrading.AutomationOrderStatus.NO_TRADE;
        }
    // . . .
```

In addition to the above, the `DomfiTrading::checkNoPendingTrigger()` will return false until the `triggerTimeout` expires:

```
    function checkNoPendingTrigger(
        address trader,
        uint16 pairIndex,
        uint8 index,
        IDomfiTradingStorage.LimitOrder orderType
    ) public view returns (bool) {
        uint256 triggerBlock =
IDomfiTradingStorage(registry.getContractAddress("tradingStorage"))
            .orderTriggerBlock(trader, pairIndex, index, orderType);

        if (
            triggerBlock == 0
                || (triggerBlock > 0 && ChainUtils.getBlockNumber() - triggerBlock >=
triggerTimeout)        <<@
        ) {
            return true;
        }
        return false;
    }
```

Hence, it will lead to a temporary blockage of the `closeTradeMarket()`, `topUpCollateral()`, and `removeCollateral()` functions, which can be crucial when a user is taking multiple trades.

Recommendations

It is recommended to add the `unregisterTrigger()` call.

```
        if (cancelReason != CancelReason.NONE) {
            emit RemoveCollateralRejected(
                a.orderId,
                tradeInfo.tradeId,
                request.trader,
                request.pairIndex,
                request.removeAmount,
                cancelReason
            );

            tradingStorage.unregisterPendingRemoveCollateral(a.orderId);
+            tradingStorage.unregisterTrigger(
+               request.trader,
+               request.pairIndex,
+               request.index,
+               IDomfiTradingStorage.LimitOrder.REMOVE_COLLATERAL
            );
            return;
        }
```

[M-03] Closing fee is calculated after updating the open interest

Severity

Impact: Low

Likelihood: High

Description

During a `DomfiTradingCallbacks::unregisterTrade()` call, the `DomfiTradingStorage::unregisterTrade()` updates the open interest first as per the direction of the trade via `DomfiTradingStorage::updateOpenInterest()` :

```
function updateOpenInterest(uint16 pairIndex, uint256 oiNotional, bool open, bool long)
    private
{
    uint256 index = long ? uint256(OpenInterest.LONG) : uint256(OpenInterest.SHORT);
    uint256[3] storage o = openInterest[pairIndex];
    uint256 newOi = open ? o[index] + oiNotional : o[index] - oiNotional;
    o[index] = newOi;

    emit OpenInterestUpdated(pairIndex, index, newOi);
}
```

However, the current logical implementation unregisters the trade before applying the closing fee, which essentially uses a post-update open interest for the fee calculation:

```
function unregisterTrade(
    uint256 orderId,
    uint256 tradeId,
    IDomfiTradingStorage.Trade memory trade,
    uint256 usdcSentToTrader,
    uint256 liquidationFee, // PRECISION_6
    uint256 collateralToClose, // PRECISION_6
    uint256 latestPrice
) private {
    // . . .
    // 2. Unregister trade
    tradingStorage.unregisterTrade(
        trade.trader, trade.pairIndex, trade.index, collateralToClose                 <<@
    );

    // 3. Transfer any liquidation or closing fees
    uint256 closingFee = 0;

    if (liquidationFee > 0) {
        tradingStorage.transferUsdc(address(tradingStorage), address(this), liquidationFee);
        vault.distributeReward(liquidationFee);
        emit VaultLiqFeeCharged(orderId, tradeId, trade.trader, liquidationFee);
    } else {
        // only charge closing fee if not a liquidation
        if (usdcSentToTrader > 0) {
                uint256 size = collateralToClose.mulDiv(trade.leverage, 100,
Math.Rounding.Ceil);

                (uint256 reward, uint256 vaultReward) =
tradingStorage.handleClosingFees(           <<@
                    trade.pairIndex, latestPrice, size, trade.leverage, trade.buy
                );

                closingFee += reward;
                usdcSentToTrader -= reward;

                emit DevFeeCharged(tradeId, trade.trader, reward);

                if (vaultReward > 0) {
                    tradingStorage.transferUsdc(address(tradingStorage), address(this),
vaultReward);
                    vault.distributeReward(vaultReward);
                    closingFee += vaultReward;
                    usdcSentToTrader -= vaultReward;
                    emit VaultClosingFeeCharged(tradeId, trade.trader, vaultReward);
                }
            }
        }
        // . . .
    }
```

We can determine that closing a trade is equivalent to taking a position against the current direction; charging fees after the OI update would undermine the skew created by the position.

Recommendations

It is recommended to use the pre-close OI for calculating the closing fee:

```
    function handleClosingFees(
        uint16 pairIndex,
        uint256 latestPrice,
        uint256 leveragedPositionSize,
        uint32 leverage,
        bool isBuy
    ) external onlyCallbacks returns (uint256 devFee, uint256 vaultFee) {
        uint256 oiLong = (openInterest[pairIndex][uint256(OpenInterest.LONG)] * latestPrice)
            / PRECISION_18 / 1e12;
        uint256 oiShort = (openInterest[pairIndex][uint256(OpenInterest.SHORT)] * latestPrice)
            / PRECISION_18 / 1e12;

        // oiDelta AFTER the close
        int256 oiDeltaAfter = oiLong.toInt256() - oiShort.toInt256();

        // Closing a long is a short trade; closing a short is a long trade
        int256 tradeSizeSigned =
            isBuy ? -leveragedPositionSize.toInt256() : leveragedPositionSize.toInt256();

        // Reconstruct oiDelta BEFORE the close
        int256 oiDeltaBefore = oiDeltaAfter - tradeSizeSigned;
        (devFee, vaultFee) =
            IDomfiPairInfos(registry.getContractAddress("pairInfos")).getClosingFee(
                pairIndex,
                tradeSizeSigned,
                leverage,
                oiDeltaBefore
            );

        devFees += devFee;
    }
```

[M-04] Donation attack through `distributeReward` to inflate share price

Severity

Impact: Medium

Likelihood: Medium

Description

Contract DomfiVault converts assets to shares based on the `shareToAssetsPrice` variable, and the default value of the `shareToAssetsPrice` is 1e18, but the code multiplies and divides by `PRECISION_18` to remove the 18 digit precision and so shares and asset decimals are the same:

```
    function _convertToShares(uint256 assets, Math.Rounding rounding)
        internal
        view
        override
        returns (uint256)
    {
        return assets.mulDiv(PRECISION_18, shareToAssetsPrice, rounding);
    }

    function _convertToAssets(uint256 shares, Math.Rounding rounding)
        internal
        view
        override
        returns (uint256)
    {
        if (shares == type(uint256).max && shareToAssetsPrice >= PRECISION_18) {
            return shares;
        }
        return shares.mulDiv(shareToAssetsPrice, PRECISION_18, rounding);
    }
```

The value of the `shareToAssetsPrice` is updated in the function `updateShareToAssetsPrice()`, which is based on `accRewardsPerToken` and `accPnlPerTokenUsed`:

```
function updateShareToAssetsPrice() private {
    shareToAssetsPrice = maxAccPnlPerToken()

        - (accPnlPerTokenUsed > 0 ? uint256(accPnlPerTokenUsed) : uint256(0));

    emit ShareToAssetsPriceUpdated(shareToAssetsPrice);
}

function maxAccPnlPerToken() public view returns (uint256) {
    return accRewardsPerToken + PRECISION_18;
}
```

Function `distributeReward()` is publicly callable and increases `accRewardsPerToken` and calls `updateShareToAssetsPrice()` to update the share price:

```
function distributeReward(uint256 assets) external {
    address sender = _msgSender();
    SafeERC20.safeTransferFrom(_assetIERC20(), sender, address(this), assets);

    accRewardsPerToken += assets * PRECISION_18 / totalSupply();
    updateShareToAssetsPrice();

    emit RewardDistributed(sender, assets, accRewardsPerToken);
}
```

So when the function `distributeReward()` is called, the share price increases, and the increase depends on the amount of assets and the total supply. An attacker can use this function to inflate the share price by this:

1. When the contract is deployed recently, the attacker will deposit 1 wai asset and mint 1 wei share.
2. Then the attacker will call `distributeReward()` and will distribute 100e18 asset, and it will increase the share price by a factor of 100e18.
3. Now the `shareToAssetsPrice` will be so high, and users will lose funds up to 100e18 tokens in deposits because of a rounding error.

Recommendations

Add an extra decimal for the share value or limit calling `distributeReward()` only to system contracts.

[M-05] No slippage control for withdraw and redeem in the vault

Severity

Impact: Low

Likelihood: High

Description

The Vault's share price can change when `updateShareToAssetsPrice()` is called, which changes the share price based on `accPnlPerTokenUsed` :

```
function updateShareToAssetsPrice() private {
    shareToAssetsPrice = maxAccPnlPerToken()

        - (accPnlPerTokenUsed > 0 ? uint256(accPnlPerTokenUsed) : uint256(0));

    emit ShareToAssetsPriceUpdated(shareToAssetsPrice);
}
```

Function `unlockDeposit()` is callable by users to finalize their locked deposits, and it updates the `accPnlPerTokenUsed` and calls `updateShareToAssetsPrice()` :

```
    // slither-disable-next-line reentrancy-benign
    lockedDepositNft.burn(depositId);

    accPnlPerTokenUsed += accPnlDelta;
    updateShareToAssetsPrice();
```

So it is possible for the share price to change by other users' interactions. The issue is that functions redeem/withdraw and deposit/mint do not allow users to set slippage, and users may receive an undesired amount of shares or tokens for their transaction if the share price changes before the users' transaction execution.

Recommendations

Allow users to set the slippage parameter and check the share and asset amount based on the slippage.

[M-06] Function `removePendingAutomationOrder` would revert

Severity

Impact: Low

Likelihood: High

Description

Code registers an automation order when the function `executeAutomationOrder()` is called, and the function `removePendingAutomationOrder()` is supposed to remove the pending automation order and triggers:

```
    if (!checkNoPendingTriggers(trader, pairIndex, index)) {
        return IDomfiTrading.AutomationOrderStatus.PENDING_TRIGGER;
    }
```

```
        tradingStorage.unregisterTrigger(trader, pairIndex, index, limitType);
        tradingStorage.unregisterPendingAutomationOrder(orderId);
```

The issue is that in the `removePendingAutomationOrder()` code calls `checkNoPendingTriggers()` to make sure there are no triggers, but the pending automation order is coupled with a pending trigger, and if there is a pending automation order, then a trigger exists for that order too, as code unregisters the trigger in the next lines. So this function is always reverted as pending automation orders will have a pending trigger too.

Recommendations

Do not check for pending triggers when removing the automation order.

[M-07] `executeAutomationOrder()` lacks order time checks and enables frontrunning

Severity

Impact: Low

Likelihood: High

Description

When the price reaches a certain level, then an automation order can be executed, and the off-chain bot calls `executeAutomationOrder()` with the index of that order to create a trigger for that order and execute the order in the callback contract:

```
    function executeAutomationOrder(
        IDomfiTradingStorage.LimitOrder orderType,
        address trader,
        uint16 pairIndex,
        uint8 index,
        uint256 validator
    )
        external
        onlyTradesUpKeep
        notDone
        pairIndexListed(pairIndex)
        returns (IDomfiTrading.AutomationOrderStatus, uint256 orderId)
    {
        IDomfiTradingStorage tradingStorage =
            IDomfiTradingStorage(registry.getContractAddress("tradingStorage"));

        if (orderType == IDomfiTradingStorage.LimitOrder.OPEN) {
            if (!tradingStorage.hasOpenLimitOrder(trader, pairIndex, index)) {
                return (IDomfiTrading.AutomationOrderStatus.NO_LIMIT, orderId);
            }
            isNotPaused();
        } else {
```

```
--snip--

        tradingStorage.storePendingAutomationOrder(
            IDomfiTradingStorage.PendingAutomationOrder(trader, pairIndex, index, orderType),
            orderId
        );

        tradingStorage.setTrigger(trader, pairIndex, index, orderType);
```

The issue is that the off-chain bot specifies the trade/order index, and orders and positions can be replaced or changed before the `executeAutomationOrder()` execution, and the code does not check order/trade creation and modification time in the `executeAutomationOrder()`. This is one scenario that an attacker can use to extract value with limit orders: 1- Attacker creates a limit order at price 100. 2- At time N the price reaches 100. 3- The off-chain bot calls `executeAutomationOrder()` a few seconds later to execute the limit order, and the price is 95 at the current time. 4- User monitors the price movements, will see the price going down, and will front-run the `executeAutomationOrder()` and will call `updateOpenLimitOrder()` to change the target price for his order to 95. 5- As a result, the user order will be executed at a price of 95.

Because their delay between the time the price reaches the target level and the off-chain bot calls the `executeAutomationOrder()`, the user will always have enough time to watch future price movements and update or replace his trade or order to extract value.

Recommendations

Code should check that the order and position last update time and creation time are before the target `timestamp` that is defined in the `executeAutomationOrder()` with the `validator` variable.

[M-08] Minimum position size can be bypassed in `openTrade` function

Severity

Impact: Low

Likelihood: High

Description

When users call `openTrade` to open a new trade code, it checks the position size to make sure it is bigger than the minimum limit:

```
        if ((t.collateral * t.leverage) / 100 < pairsStorage.pairMinLevPos(t.pairIndex)) {
            revert BelowMinLevPos();
        }
```

The issue is that the code uses pre-fee collateral, and actual collateral will be lower because fees will be deducted from it. As a result, it would be possible to create positions that their size is less than the minimum limit.

Recommendations

Perform the check with post-fee collateral amount and the highest level of fee (taker fee) that can be applied to estimate the fee.

[M-09] Execute `executeAutomationOrder()` ignores SL/TP last update timestamp

Severity

Impact: Low

Likelihood: High

Description

Function `executeAutomationOrder()` is called when the user position SL/TP is reached. The off-chain bot calls this function with the timestamp that the price reaches the SL/TP and the code sets a trigger to request the price at that timestamp and perform the automatic order execution:

```
            if (orderType == IDomfiTradingStorage.LimitOrder.SL && t.sl == 0) {
                return (IDomfiTrading.AutomationOrderStatus.NO_SL, orderId);
            }

            if (orderType == IDomfiTradingStorage.LimitOrder.TP && t.tp == 0) {
                return (IDomfiTrading.AutomationOrderStatus.NO_TP, orderId);
            }

---snip
        if (requirePrice) {
            // increment `orderId` and store it while making a price request
            orderId = priceRouter.getPrice(pairIndex, upkeepOrderType, validator);
        } else {
            // increment `orderId` and store it without requesting a price update
            orderId = priceRouter.storeOrder(pairIndex, upkeepOrderType, validator);
        }

        tradingStorage.storePendingAutomationOrder(
            IDomfiTradingStorage.PendingAutomationOrder(trader, pairIndex, index, orderType),
            orderId
        );
```

The issue is that the code does not check the last update timestamp of SL/TP to be before the requested price timestamp and as a result users can watch the mempool and market prices and when they see the price change is in favor of not executing their SL/TP or they could

benefit from another SL/TP value then they will front run the `executeAutomationOrder()` transaction and change the SL/TP. This is one scenario:
1- User TP is at price 100 in timestamp X.
2- Price reaches 105, and there is a call to `executeAutomationOrder()` for user TP order execution in timestamp X.
3- User sees this transaction in the mempool and front-runs it and changes TP to 104.
4- As a result, the user position will be closed at 104, and the user will get more profit.

As the off-chain bot has some delays, the user can always change TP/SL after the target price request timestamp and extract more value without risk.

Recommendations

Function `executeAutomationOrder()` must only allow automatic order execution if the requested price timestamp is after the last SL/TP update timestamp:

```
            if (
                orderType == IDomfiTradingStorage.LimitOrder.SL
                    && (t.sl == 0 || (t.sl != 0 && tradeInfo.slLastUpdated > validator)) //
assuming validator is timestamp
            ) {
                return (IDomfiTrading.AutomationOrderStatus.NO_SL, orderId);
            }
            if (
                orderType == IDomfiTradingStorage.LimitOrder.TP
                    && (t.tp == 0 || (t.tp != 0 && tradeInfo.tpLastUpdated > validator)) //
assuming validator is timestamp
            ) {
                return (IDomfiTrading.AutomationOrderStatus.NO_TP, orderId);
            }
```

[M-10] Trade validation performed with pre-fee collateral

Severity

Impact: Medium

Likelihood: Medium

Description

When opening a trade, the protocol validates exposure limits before deducting opening fees from the trader's collateral.
In `openTradeMarketCallback()`, the validation checks are performed using the original collateral amount:

```
cancelReason = TradingCallbacksLib.getOpenTradeMarketCancelReason(
    isPaused,
    wantedPrice,
    slippageP,
    uint192(a.price),
    trade,  // Contains pre-fee collateral
    priceImpactP,
    IDomfiPairInfos(registry.getContractAddress("pairInfos")),
    IDomfiPairsStorage(registry.getContractAddress("pairsStorage")),
    IDomfiTradingStorage(registry.getContractAddress("tradingStorage"))
);
```

Only after validation passes does the protocol deduct fees in `registerTrade()`. The exposure limit validation in `withinExposureLimits()` uses the pre-fee collateral amount:

```
function withinExposureLimits(
    uint16 pairIndex,
    bool buy,
    uint256 collateral,  // Pre-fee amount
    uint32 leverage,
    uint256 price,
    IDomfiPairsStorage pairsStorage,
    IDomfiTradingStorage tradingStorage
) public view returns (bool) {
    uint256 oiType = buy
        ? uint256(IDomfiTradingStorage.OpenInterest.LONG)
        : uint256(IDomfiTradingStorage.OpenInterest.SHORT);

    return tradingStorage.openInterest(pairIndex, oiType) * price / PRECISION_18 / 1e12
        + collateral * leverage / 100  // Validation uses pre-fee collateral
        <= tradingStorage.openInterest(pairIndex,
uint256(IDomfiTradingStorage.OpenInterest.MAX))
        && pairsStorage.groupCollateral(pairIndex, buy) + collateral  // Also pre-fee
            <= pairsStorage.groupMaxCollateral(pairIndex);
}
```

This creates a discrepancy where the validation is performed against a higher collateral amount than what is actually stored and used for the trade.

The same issue occurs in `executeAutomationOpenOrderCallback()`, where validation happens before fee deduction.

Recommendations

Consider calculating and deducting fees before performing the validations.

[M-11] Inconsistent price usage between OI validation and storage

Severity

Impact: Medium

Likelihood: Medium

Description

Open interest is stored in asset units rather than collateral units. In `DomfiTradingStorage.sol`, OI values are converted to collateral units by multiplying by the current price:

```
uint256 oiLong = (openInterest[pairIndex][uint256(OpenInterest.LONG)] * latestPrice) /
PRECISION_18 / 1e12;
uint256 oiShort = (openInterest[pairIndex][uint256(OpenInterest.SHORT)] * latestPrice) /
PRECISION_18 / 1e12;
```

Similarly, in `withinExposureLimits()`, the protocol correctly converts stored asset open interest to collateral units using the oracle price for validation:

```
return tradingStorage.openInterest(pairIndex, oiType) * price / PRECISION_18 / 1e12
    + collateral * leverage / 100
    <= tradingStorage.openInterest(pairIndex, uint256(IDomfiTradingStorage.OpenInterest.MAX))
```

During trade validation in `getTradePriceImpact()`, the new trade's notional is calculated using the oracle price, which is the raw price from the price feed before any price impact adjustments are applied:

```
uint256 oiNotional = (collateral * PRECISION_12 * leverage / 100) * PRECISION_18 /
uint192(price);
```

In this calculation, the `price` parameter represents the oracle price passed from `a.price` in the callback function. This calculated `oiNotional` is then used to compute the hypothetical post-trade open interest state, which determines the price impact that traders will experience. The calculation converts the trade's collateral value to an asset quantity by dividing by the oracle price, yielding a result in asset units (18 decimals).

However, when the trade is actually registered and stored in `registerTrade()`, the notional is calculated using `trade.openPrice`, which is the price after impact has been applied:

```
tradingStorage.storeTrade(
    trade,
    IDomfiTradingStorage.TradeInfo(
        tradeId,
        trade.collateral * uint256(1e12) * trade.leverage / PRECISION_2 * PRECISION_18
            / trade.openPrice,  // Uses post-impact price
        trade.leverage,
        currBlock,
        currBlock,
        currBlock,
        false
    )
);
```

The `trade.openPrice` field was set earlier in the execution flow to the price after impact:

```
(uint256 priceImpactP, uint256 priceAfterImpact) = TradingCallbacksLib.getTradePriceImpact(
    a.price, trade.collateral, trade.leverage, tradingStorage, trade.pairIndex, true, trade.buy
);
trade.openPrice = priceAfterImpact.toUint192();
```

This creates a fundamental inconsistency where the price used for validation differs from the price used for storage, even though both calculations are meant to represent the same conceptual quantity of assets being traded.

This results in wrong price impact calculations, affecting trades' open prices.

Recommendations

Consider changing the calculation in `getTradePriceImpact` to use the collateral units for OI calculations, similar to `withinExposureLimits()` and `handleOpening/ClosingFees`. This would ensure consistency in how open interest is calculated and validated throughout the trade lifecycle. Something similar to:

```
(uint256 oiLong, uint256 oiShort,,,) = tradingStorage.getPairOpenInterestInfo(pairIndex);
oiLong = oiLong * uint192(price) / PRECISION_18 / 1e12;
oiShort = oiShort * uint192(price) / PRECISION_18 / 1e12;
uint256 oiNotional = collateral * leverage / 100;
```

[M-12] Automation close order validation uses an inconsistent price

Severity

Impact: Medium

Likelihood: Medium

Description

When executing automation close orders (liquidations, stop losses, and take profits), the protocol calculates profit and validates the order in two steps. However, for liquidations and stop losses, these steps use inconsistent prices, leading to potential validation errors.

In `executeAutomationCloseOrderCallback`, for liquidations and stop losses (`isMarketPrice = true`), the profit is calculated using the raw oracle price `a.price`:

```
bool isMarketPrice = orderType == IDomfiTradingStorage.LimitOrder.LIQ
    || orderType == IDomfiTradingStorage.LimitOrder.SL;

(int256 profitP,) = TradingCallbacksLib.currentPercentProfit(
    t.openPrice.toInt256(),
    isMarketPrice ? a.price : priceAfterImpact.toInt256(), // Uses a.price for liquidations
    t.buy,
    int32(t.leverage),
    int32(i.initialLeverage)
);
```

However, the validation check always uses `priceAfterImpact` regardless of the order type:

```
cancelReason = TradingCallbacksLib.getAutomationCloseOrderCancelReason(
    orderType, t, priceAfterImpact, usdcSentToTrader // Always uses priceAfterImpact
);
```

This inconsistency means that:

1. The `usdcSentToTrader` is calculated based on `a.price` (for liquidations/stop losses).
2. But the validation uses `priceAfterImpact` to check if the order should execute.

This can lead to incorrect order execution decisions.

Recommendations

Consider using `isMarketPrice` to determine which price to pass to `getAutomationCloseOrderCancelReason`:

```
cancelReason = TradingCallbacksLib.getAutomationCloseOrderCancelReason(
    orderType, t, isMarketPrice ? a.price : priceAfterImpact, usdcSentToTrader
);
```

[M-13] Timed-out collateral orders not cleaned when trigger pending

Severity

Impact: Medium

Likelihood: Medium

Description

The `removePendingRemoveCollateral` function is used by TradesUpKeep to clean up timed-out remove collateral requests. However, it checks for all pending trigger types instead of only checking the `REMOVE_COLLATERAL` trigger:

```
function removePendingRemoveCollateral(uint256 orderId)
    external
    onlyTradesUpKeep
    notDone
    returns (IDomfiTrading.AutomationOrderStatus)
{
    // ... snip ...

    // Checks ALL triggers (TP, SL, LIQ, REMOVE_COLLATERAL)
    if (!checkNoPendingTriggers(pending.trader, pending.pairIndex, pending.index)) {
        return IDomfiTrading.AutomationOrderStatus.PENDING_TRIGGER;
    }
    // ... snip ...
}
```

This creates a situation where if a trade has any other pending trigger (TP, SL, or LIQ) that has not timed out yet, the timed-out `REMOVE_COLLATERAL` order cannot be cleaned up until those other triggers are resolved or timed out.

This can lead to unnecessary delays in cleaning up stale remove collateral requests.

Recommendations

Consider checking only the `REMOVE_COLLATERAL` trigger timeout instead of all triggers.

[M-14] No slippage protection on market close orders

Severity

Impact: Medium

Likelihood: Medium

Description

When traders initiate market close orders, they have no ability to specify slippage protection. Unlike `openTrade`, which accepts a `slippageP` parameter, `closeTradeMarket` does not, and hardcodes slippage to 0 when storing the pending order.

`closeTradeMarket` provides no such protection:

```
function closeTradeMarket(uint16 pairIndex, uint8 index, uint16 closePercentage)
    external
    payable
    notDone
{
    // No slippageP parameter at all
    // ...

    tradingStorage.storePendingMarketOrder(
        IDomfiTradingStorage.PendingMarketOrder(
            0,
            0,
            0, // Hardcoded to 0 - no slippage protection
            IDomfiTradingStorage.Trade(0, 0, 0, 0, sender, 0, pairIndex, index, t.buy),
            closePercentage
        ),
        orderId,
        false
    );
}
```

Because `slippageP` is hardcoded to 0, `closeTradeMarketCallback` cannot validate slippage.

This means traders closing positions are completely exposed to unfavorable price movements.

Recommendations

Consider adding slippage protection to `closeTradeMarket`.

[M-15] Limit order execution does not validate SL TP placement

Severity

Impact: Medium

Likelihood: Medium

Description

`getAutomationOpenOrderCancelReason`, which is called in open limit order callback, validates that the limit trigger is hit, exposure limits, max negative PnL on open, and max leverage; but does not validate SL/TP constraints.

```
if (isNotHit) return CancelReason.NOT_HIT;
if (!withinExposureLimits(...)) return CancelReason.EXPOSURE_LIMITS;
if (!withinMaxNegativePnlOnOpenP(...)) return CancelReason.PRICE_IMPACT;
if (!withinMaxLeverage(...)) return CancelReason.MAX_LEVERAGE;
return CancelReason.NONE;
```

This allows an automated limit order to be executed into a trade whose SL/TP is immediately invalid relative to the actual execution price, potentially leading to instant closure / instant SL (or nonsensical TP).

Recommendations

Consider adding SL/TP validation in `getAutomationOpenOrderCancelReason`:

```
// Check if TP is reached
if (o.tp != 0 && (o.buy ? priceAfterImpact >= o.tp : priceAfterImpact <= o.tp)) {
    return IOstiumTradingCallbacks.CancelReason.TP_REACHED;
}

// Check if SL is reached
if (o.sl != 0 && (o.buy ? priceAfterImpact <= o.sl : priceAfterImpact >= o.sl)) {
    return IOstiumTradingCallbacks.CancelReason.SL_REACHED;
}
```

[M-16] Price impact miscalculation on partial market close

Severity

Impact: Medium

Likelihood: Medium

Description

When closing a trade via `closeTradeMarketCallback`, the protocol allows users to partially close an open position by specifying `closePercentage`. The callback correctly applies this percentage when calculating:

- Collateral to close.
- Trade value.
- PnL distribution.
- Open interest reduction.

However, during price impact calculation, the callback invokes `TradingCallbacksLib.getTradePriceImpact` using the **full position collateral and leverage** taken from the trade, instead of the **portion of the position being closed**.

```
(uint256 priceImpactP, uint256 priceAfterImpact) =
    TradingCallbacksLib.getTradePriceImpact(
        a.price,
        trade.collateral,
        trade.leverage,
        tradingStorage,
        trade.pairIndex,
        false,
        trade.buy
    );
```

This inconsistency causes price impact to be exaggerated, potentially leading to user losses, incorrect PnL, or unfair execution.

Recommendations

Consider scaling the collateral by `closePercentage`, similar to how collateral and open interest are handled elsewhere in the callback.

[M-17] Pending automation order for OPEN limit can be removed before timeout

Severity

Impact: Medium

Likelihood: Medium

Description

`removePendingAutomationOrder` allows a pending OPEN limit automation order to be removed as long as no other triggers (TP/SL/LIQ/REMOVE_COLLATERAL) are pending for the trade:

```
if (!checkNoPendingTriggers(trader, pairIndex, index)) {
    return IDomfiTrading.AutomationOrderStatus.PENDING_TRIGGER;
}
```

However, this check does not enforce the trigger timeout for the OPEN limit order itself. As a result, an OPEN limit order's automation trigger can be removed even if it was just registered and has not yet passed the required `triggerTimeout`.

This allows the `TradesUpKeep` flow to prematurely invalidate an OPEN limit execution attempt before the system-defined timeout window has elapsed.

Recommendations

Consider requiring the OPEN limit trigger itself to have passed the timeout before allowing removal:

```
if (
    !checkNoPendingTriggers(trader, pairIndex, index) ||
    !checkNoPendingTrigger(trader, pairIndex, index, limitType)
) {
    return IDomfiTrading.AutomationOrderStatus.PENDING_TRIGGER;
}
```

[M-18] Uncapped closing fees can cause underflow and block trade closure

Severity

Impact: Medium

Likelihood: Medium

Description

When trades are closed, the protocol charges different kinds of fees (funding, rollover, and closing). Funding fees are fees that are transferred between longs and shorts, so closed trades could benefit from this. Rollover fees are charged depending on the length of time the trade was opened for, handled in `getTradeRolloverFeePure`.

```
uint256 rolloverFee = (
    (endAccRolloverFeesPerCollateral - accRolloverFeesPerCollateral) * collateral * leverage
) / PRECISION_18 / PRECISION_2
```

The longer the trade is open, the more rollover fees accumulate. Closing fees are also calculated depending on configuration plus open trades delta, handled in `_getBaseClosingFee`.

The gist is that over time, the fees paid when closing will continue to increase.

On the other hand, when closing, the amount sent to the trader is calculated by subtracting the different fees:

```
// only charge closing fee if not a liquidation
if (usdcSentToTrader > 0) {
    uint256 size = collateralToClose.mulDiv(trade.leverage, 100, Math.Rounding.Ceil);

    (uint256 reward, uint256 vaultReward) = tradingStorage.handleClosingFees(
        trade.pairIndex, latestPrice, size, trade.leverage, trade.buy
    );

    closingFee += reward;
    usdcSentToTrader -= reward; // <--- here

    emit DevFeeCharged(tradeId, trade.trader, reward);

    if (vaultReward > 0) {
        tradingStorage.transferUsdc(address(tradingStorage), address(this), vaultReward);
        vault.distributeReward(vaultReward);
        closingFee += vaultReward;
        usdcSentToTrader -= vaultReward; // <--- here
        emit VaultClosingFeeCharged(tradeId, trade.trader, vaultReward);
    }
}

// 4. Transfer USDC from the vault to the trader in case the position has a positive PnL after
fees, otherwise
// transfer the negative PnL + fees worth of USDC from the trader to the vault
uint256 usdcLeftInStorage = collateralToClose - liquidationFee - closingFee; // <--- here
```

However, if these fees grow large enough, the subtraction could underflow, mainly because fees do not depend on the closing trade's value.

This blocks traders from closing their trades.

Proof of Concept: https://gist.github.com/0xAlix2/54fe437d5e3d1a74927c8b9b9189f7d4

Recommendations

Consider capping closing fees to a configurable percentage of the closed trade value, or simply taking all the value as a closing fee and not returning anything to the user (this could be a bit confusing for the traders).

[M-19] `updateAccPnlPerTokenUsed()` only applies positive unrealized PnL

Severity

Impact: Medium

Likelihood: Medium

Description

When an epoch ends and a new epoch starts, the function `updateAccPnlPerTokenUsed()` is executed, and it updates the share price. This function is called by the function `startNewEpoch()`, which sends unrealized accumulated profit and loss at the start of the epoch and at the end of the epoch as input for the function `updateAccPnlPerTokenUsed()`:

```
        uint256 currentEpochPositiveOpenPnl = vault.currentEpochPositiveOpenPnl();

        uint256 finalNewEpochPositiveOpenPnl = vault.updateAccPnlPerTokenUsed(
            currentEpochPositiveOpenPnl, newEpochOpenPnl > 0 ? uint256(newEpochOpenPnl) : 0
        );
```

The issue is that this function only sends the positive unrealized PnL for the vault, and the vault only applies the positive traders' PnL in the vault's share price. It means if traders have unrealized losses, then the vault share price will not be updated even after the epoch ends. This would create situations in which users can extract value from the vault. For example:

1. Traders made realized profits, and the vault of `accPnlPerToken` is positive, and the share price decreased (if traders make a realized loss, then the share price will increase).
2. UserA has a trade that has a big unrealized loss, and the epoch's net unrealized PnL is negative.
3. In function `startNewEpoch()`, because `newEpochOpenPnl < 0`, the code will not send the unrealized loss to the vault, and the vault share price will not be increased.
4. To capture some of the value that LPs receive from the loss, UserA would deposit a large amount of assets into the vault (for example, owning 50% of the vault's share) and then close his trade and make a realized loss.
5. Now the code would increase the vault's share price, and 50% of the gained yield will be received by UserA himself.
6. Then UserA would withdraw his funds.

UserA can create a big delta neutral position to perform this attack too. As a result of this issue, attackers have opportunities to extract value from the vault when there is a position with loss and the accumulated PnL in the vault is positive.

Recommendations

Always update the vault's share price based on accumulated unrealized profit and loss.

[M-20] Code does not reset `orderTriggerBlock` when updating trades

Severity

Impact: Medium

Likelihood: Medium

Description

Code uses the variable `orderTriggerBlock` to track the events triggered for an order or trade. To perform an operation on an order or trade, code checks that there is no current ongoing trigger by calling the `checkNoPendingTrigger()` function:

```
    function checkNoPendingTrigger(
        IOstiumTradingStorage storageT,
        address trader,
        uint16 pairIndex,
        uint8 index,
        IOstiumTradingStorage.LimitOrder orderType,
        uint256 triggerTimeout
    ) public view returns (bool) {
        uint256 triggerBlock = storageT.orderTriggerBlock(trader, pairIndex, index, orderType);

        if (triggerBlock == 0 || (triggerBlock > 0 && ChainUtils.getBlockNumber() -
triggerBlock >= triggerTimeout)) {
            return true;
        }
        return false;
    }
```

The issue is that when removing a trade, code does not reset the value of the `orderTriggerBlock[]` for that specific index. And code may reuse the index to store future trades. As a result, the next trade will have a value in `orderTriggerBlock` and some operations will fail for that trade. For example: 1- UserA has an open trade and calls `removeCollateral`, and code creates a remove collateral trigger. 2- Next, the user trade is liquidated, but the value of `orderTriggerBlock[]` is not removed for that specific trade's index. 3- UserA opens another trade, and it is stored in the same trade index position. 4- Now the new trade has the value of `orderTriggerBlock[]` for removing collateral, and it will not be possible to perform that action for some time.

This issue can happen with different triggers, and it would create a failure if users try to do high frequency trading.

Recommendations

In function `unregisterTrade()`, reset all the trigger types values for that trade in variable `orderTriggerBlock`

[M-21] Using average for unrealized PnL settlement gives opportunity to extract value

Severity

Impact: Low

Likelihood: High

Description

When an epoch ends, the code settles the unrealized profit and loss in the vault. It uses the average of the snapshot of the unrealized PnL during the epoch:

```
        int256 newEpochOpenPnl =
            nextEpochValues.length >= requestsCount ? average(nextEpochValues) :
currentEpochPositiveOpenPnl.toInt256();

        uint256 finalNewEpochPositiveOpenPnl = vault.updateAccPnlPerTokenUsed(
            currentEpochPositiveOpenPnl, newEpochOpenPnl > 0 ? uint256(newEpochOpenPnl) : 0
        );
```

The issue is that there may be a high difference between the current net PnL and its average value, especially if the price changes dramatically near the end of the epoch. As a result, in the next epoch, users would have the opportunity to deposit/withdraw while this gain/loss is not included in the vault's share price and benefit more than other LP providers.

Recommendations

Do not use average or settle the deposits/withdraw with the share price of the next epoch.

[M-22] Discrepancy for `isLiquidated` in trade and automation callbacks

Severity

Impact: Medium

Likelihood: Medium

Description

According to the docs, the liquidation trigger should be calculated based on the market price and executed with price after impact. The `closeTradeMarketCallback()` follows this behavior:

```
                (
                    TradingCallbacksLib.TradeValueResult memory tvResult,
                    TradingCallbacksLib.PriceImpactResult memory piResult
                ) = TradingCallbacksLib.getTradeAndPriceData(
                    a,
                    t,
                    pairInfos,
                    i.initialLeverage,

IOstiumPairsStorage(registry.getContractAddress('pairsStorage')).pairMaxLeverage(t.pairIndex),
                    collateralToClose,
                    true
                );

                bool isLiquidated = tvResult.tradeValue < tvResult.liqMarginValue;
```

But in the function `executeAutomationCloseOrderCallback()`, the code recalculates the `isLiquidated` based on the trade value of price after impact:

```
                if (isMarketPrice) {
                    (tvResult.profitP,) = TradingCallbacksLib.currentPercentProfit(
                        t.openPrice.toInt256(),
                        piResult.priceAfterImpact.toInt256(),
                        t.buy,
                        int32(t.leverage),
                        int32(i.initialLeverage)
                    );
                    tvResult.tradeValue = pairInfos.getTradeValuePure(
                        t.collateral, tvResult.profitP, tvResult.rolloverFees,
tvResult.fundingFees
                    );

                    isLiquidated = tvResult.tradeValue < tvResult.liqMarginValue;
                }
```

As a result, if the user has SL close to the liquidation price in order to avoid the liquidation, if the SL gets triggered, then because of this second `isLiquidated` calculation based on the `priceAfterImpact`, the position will be liquidated. Also, this happens in the `getHandleRemoveCollateralCancelReason()` function too:

```
        (int256 profitP, int256 maxPnlP) = currentPercentProfit(
            trade.openPrice.toInt256(),
            result.priceAfterImpact.toInt256(),
            trade.buy,
            int32(trade.leverage),
            int32(initialLeverage)
        );
        uint32 maxLeverage = pairsStorage.pairMaxLeverage(trade.pairIndex);
        (uint256 tradeValue, uint256 liqMarginValue,,) = pairInfos.getTradeValue(
            trade.trader,
            trade.pairIndex,
            trade.index,
            trade.buy,
            trade.collateral,
            trade.leverage,
            profitP,
            maxLeverage
        );

        bool isLiquidated = tradeValue < liqMarginValue;
        uint256 usdcSentToTrader = isLiquidated ? 0 : tradeValue;
```

Code calculates `isLiquidated` based on the `price after impact`.

This is a discrepancy between the docs and implementation, and will cause users to pay the liquidation fee while their position is healthy according to the market price. Also, if this should be the default behavior, then users can avoid this liquidation by closing their orders with the market price when the position is about to liquidate.

Recommendations

Either fix the `closeTradeMarketCallback()` to trigger liquidation based on the price after impact, or change the `executeAutomationCloseOrderCallback()` and `getHandleRemoveCollateralCancelReason()` to not trigger liquidation with the market price.

[M-23] Attacker can front-run unrealized PnL as it takes effect at the end of epoch

Severity

Impact: Medium

Likelihood: Medium

Description

Code calculates and applies unrealized profit/loss at the start/end of the epoch in function `startNewEpoch()`:

```
        uint256 currentEpochPositiveOpenPnl = vault.currentEpochPositiveOpenPnl();

        int256 newEpochOpenPnl =
            nextEpochValues.length >= requestsCount ? average(nextEpochValues) :
currentEpochPositiveOpenPnl.toInt256();
```

```
        uint256 finalNewEpochPositiveOpenPnl = vault.updateAccPnlPerTokenUsed(
            currentEpochPositiveOpenPnl, newEpochOpenPnl > 0 ? uint256(newEpochOpenPnl) : 0
        );
```

The issue is that this will allow an attacker to front-run the unrealized profits and deposit before the epoch ends or front-run the unrealized losses and withdraw before the epoch ends (Also, the attacker or users need to have pending withdrawal requests to do this).

Recommendations

Apply unrealized loss in share price in each trade and deposit/withdraw or perform deposit/withdraw with share price as the minimum of the end of the epoch and request time (This will change the deposit/withdraw process).

[M-24] Front-running `unlockDeposit` can avoid loss distribution

Severity

Impact: Medium

Likelihood: Medium

Description

When users deposit tokens with a lock, the code gives them bonus shares, and after the lock expires, the bonus is socialized among the share token holders in function `unlockDeposit()`:

```
        int256 accPnlDelta = d.assetsDiscount.mulDiv(PRECISION_18, totalSupply(),
Math.Rounding.Ceil).toInt256();

        accPnlPerToken += accPnlDelta;
        if (accPnlPerToken > maxAccPnlPerToken().toInt256()) {
            revert NotEnoughAssets();
        }

        lockedDepositNft.burn(depositId);

        accPnlPerTokenUsed += accPnlDelta;
        updateShareToAssetsPrice();
```

The issue is that the code distributes this loss suddenly and it is unpredictable. An attacker can predict the unlock time or just front-run the unlock transaction and perform the withdrawal before the unlock to avoid this loss of socialization.

Recommendations

Either socialize the bonus/loss over time in each epoch or consider those pending bonuses for asset calculations when users perform withdrawals.

[M-25] Inconsistent price impact calculation

Severity

Impact: Low

Likelihood: High

Description

The `TradingCallbacksLib::getTradePriceImpact` helps in calculating the final price impact before registering the trade. It is calculated before the `DomfiTradingCallbacks::registerTrade()` call in `DomfiTradingCallbacks::openTradeMarketCallback()` and `DomfiTradingCallbacks::executeAutomationOpenOrderCallback()`.

However, the collateral value used to calculate the `priceAfterImpact` and `priceImpactP` is before fee deductions, but the dynamic spread is updated using the post-fee collateral:

```
    function openTradeMarketCallback(IDomfiPriceUpKeep.PriceUpKeepAnswer calldata a) external
notDone {

        // . . .
        (uint256 priceImpactP, uint256 priceAfterImpact) =
TradingCallbacksLib.getTradePriceImpact(  <<@
            a.price,
            trade.collateral,   <<@
            trade.leverage,
            tradingStorage,
            trade.pairIndex,
            true,
            trade.buy
        );

        if (cancelReason == CancelReason.NONE) {
            trade = registerTrade(a.orderId, trade, uint192(a.price), bf);           <<@
-- // trade.collateral is stored post Fee deductions
       // . . .
```

A similar flow can be seen in `executeAutomationOpenOrderCallback()`. From this observation, we can infer that subsequent pricing will be less accurate as it will depend on higher collateral.

Recommendations

It is recommended to calculate the price impact after deducting fees from the trade's collateral.

[M-26] DepositWithDiscountAndLock and mintWithDiscountAndLock missing slippage c

Severity

Impact: Medium

Likelihood: Medium

Description

`depositWithDiscountAndLock()` and `mintWithDiscountAndLock()` are used to acquire shares at a discount. Both functions have the same issue, but `depositWithDiscountAndLock()` will be used.

```
function depositWithDiscountAndLock(uint256 assets, uint32 lockDuration, address receiver)
        external
        checks(assets)
        validDiscount(lockDuration)
        returns (uint256)
    {
        uint256 simulatedAssets = assets

            * (PRECISION_18 * uint256(100) + lockDiscountP(collateralizationP(), lockDuration))
            / (PRECISION_18 * uint256(100));

        if (simulatedAssets > maxDeposit(receiver)) {
            revert AboveMaxDeposit();
        }

        return _executeDiscountAndLock(simulatedAssets, assets,
previewDeposit(simulatedAssets), lockDuration, receiver);
    }
```

`simulatedAssets` are the inflated assets that apply the discount from `lockDiscountP()`. Users deposit assets but receive an equivalent amount of shares as `simulatedAssets`, which are inflated to account for the discount. Now, let us examine `lockDiscountP()`:

```
function lockDiscountP(uint256 collatP, uint32 lockDuration) public view returns (uint256) {
        return (
            collatP <= uint16(100) * PRECISION_2
                ? uint256(maxDiscountP) * 1e16
                : collatP <= maxDiscountThresholdP
                    ? uint256(maxDiscountP) * 1e16 * (maxDiscountThresholdP - collatP)
                        / (maxDiscountThresholdP - uint16(100) * PRECISION_2)
                    : 0
        ) * lockDuration / MAX_LOCK_DURATION;
    }
```

If `collatP` is sufficiently high, meaning the protocol has high collateralization, the applied discount is reduced. This creates a problem because a user can lock their assets for a period but receive only a minimal bonus, which does not fairly compensate for the time the shares are locked, making it unfair to the user.

To better illustrate the issue, consider the following example:

1. Bob calls `depositWithDiscountAndLock()` with `amount = 1e18` and `lockDuration = MAX_LOCK_DURATION` (365 days).
2. `lockDiscountP()` calculates the discount, which is 1% because of the high `collatP`.
3. `_executeDiscountAndLock()` is executed, transferring the assets to the contract and locking the shares for 1 year.
4. Bob has locked his shares for 1 year but receives only a 1% bonus, which is insufficient. Now he must wait for the shares to unlock, and given such a small bonus, he would have preferred to simply call `deposit()`.

Recommendations

To solve the problem, implement a slippage check on the obtained discount. If the discount is lower than the user-specified minimum, the transaction should revert.

[M-27] Using max for share price calculation is wrong

Code calculates the share amount and burn those share amounts, so lower share price means more share will be burned from user and code should use min so the calculation be in favor of the pool. Use min instead of max to fix the issue.

```
        uint256 effectivePrice = Math.max(requestPrice, shareToAssetsPrice);
        shares = assets.mulDiv(PRECISION_18, effectivePrice, Math.Rounding.Floor);
        require(shares <= maxShares, "Slippage: too many shares burned");
```

[M-28] Variables `tpLastUpdated` and `slLastUpdated` should be validator

In current version variables `tpLastUpdated` and `slLastUpdated` tracks the validator state and they are compared with validator value in different part of the code. The issue is that when constructing the `TradeInfo` object code sets current block as their value. To fix this set the current publish ID as their initial value.

```
            IDomfiTradingStorage.TradeInfo(
                tradeId,
                trade.collateral * uint256(1e12) * trade.leverage / PRECISION_2 * PRECISION_18
                    / trade.openPrice,
                trade.leverage,
                currBlock,
                currBlock,
                currrBlock,
                false
            )
```

[M-29] `unregisterTrade()` shouldn't remove `LimitOrder.OPEN`

Function `unregisterTrade()` is supposed to remove triggers related to open position (trade) and trigger `LimitOrder.OPEN` is related to open limit orders. The issue is that both orders and trades is referenced by indexes and the triggers are stored in `orderTriggerBlock[trader][pairIndex][index]` and by removing the trigger `LimitOrder.OPEN` for a trade's index, it's possible to remove the trigger for the limit order which has the same index. Trades doesn't have trigger `LimitOrder.OPEN` and to fix the issue just remove the line that delate the `LimitOrder.OPEN`.

```
        delete orderTriggerBlock[trader][pairIndex][index][LimitOrder.LIQ];
        delete orderTriggerBlock[trader][pairIndex][index][LimitOrder.OPEN];
        delete orderTriggerBlock[trader][pairIndex][index][LimitOrder.RC];
        delete orderTriggerBlock[trader][pairIndex][index][LimitOrder.SL];
        delete orderTriggerBlock[trader][pairIndex][index][LimitOrder.TP];
```

Low findings

[L-01] Redundant validation in `updateOpenLimitOrder`

Description

In `updateOpenLimitOrder`, the following validation is performed:

```
if (price <= 0) {
    revert WrongParams();
}
```

However, `price` is of type `uint192`, which can never be negative.

Recommendations:

Consider replacing the condition with a strict zero check:

```
if (price == 0) {
    revert WrongParams();
}
```

[L-02] Position mutations are allowed even under pending market closure

Description

The `DomfiTrading::closeTradeMarket()` allows users to market close their position as per the `closePercentage`:

```
    function closeTradeMarket(uint16 pairIndex, uint8 index, uint16 closePercentage)
        external
        payable
        notDone
    {
      // . . .
        tradingStorage.storePendingMarketOrder(               <<@
            IDomfiTradingStorage.PendingMarketOrder(
                0,
                0,
                0,
                IDomfiTradingStorage.Trade(0, 0, 0, 0, sender, 0, pairIndex, index, t.buy),
                closePercentage
            ),
            orderId,
            false
        );

        emit MarketCloseOrderInitiated(orderId, i.tradeId, sender, pairIndex, closePercentage);
    }
```

When the function calls the `storePendingMarketOrder()`, it sets the `beingMarketClosed` to true, ensuring that the function cannot be used again to avoid re-entrancy attacks:

```
function storePendingMarketOrder(PendingMarketOrder calldata order, uint256 id, bool open)
    external
    onlyTrading
{
    pendingOrderIds[order.trade.trader].push(id);

    pendingMarketOrders[id] = order;
    pendingMarketOrders[id].blockNumber = ChainUtils.getBlockNumber();

    if (open) {
        pendingMarketOpenCount[order.trade.trader][order.trade.pairIndex]++;
    } else {
        pendingMarketCloseCount[order.trade.trader][order.trade.pairIndex]++;
        openTradesInfo[order.trade.trader][order.trade.pairIndex][order.trade.index]
            .beingMarketClosed = true;          <<@
    }
}
```

However, this does not prevent users from changing some other state, such as calling `updateTp()` / `updateSl()` / `topUpCollateral()` / `removeCollateral()` while a market close is pending. Hence, it allows mutations, which can depict unintended behavior and also lead to reverts if `removeCollateral()` was used.

It is recommended to add a `beingMarketClosed` check to all the above-mentioned functions.

[L-03] DomfiTradingStorage handleOracleFee() returns cumulative oracleFees

Description

The `DomfiTradingStorage::handleOracleFee()` is used to store the oracle fees received from the users and returns the `updatedOracleFees` :

```
function handleOracleFee(uint256 amount)
    external
    onlyTrading
    returns (uint256 updatedOracleFees)
{
    oracleFees += amount;
    updatedOracleFees = oracleFees;
}
```

However, when we look at the event emitted post calling this function in `DomfiTrading::closeTradeMarket()`, `DomfiTrading::openTrade()`, and `DomfiTrading::removeCollateral()`, we can observe that it intends to emit the oracle fees that were paid by the user instead of the current cumulative fees:

```
        SafeTransferLib.safeTransferETH(address(tradingStorage), oracleFeeOffered);
        uint256 oracleFeePaid = tradingStorage.handleOracleFee(oracleFeeOffered);     <<@
        emit OracleFeeCharged(sender, t.pairIndex, oracleFeePaid);             <<@
```

It is also non-intuitive, as once the oracle fees are claimed by the protocol, the `updatedOracleFees` would also reset, making it difficult to track what was actually paid by the user.

It is recommended to emit the `oracleFeeOffered` variable instead:

```
        SafeTransferLib.safeTransferETH(address(tradingStorage), oracleFeeOffered);
        uint256 oracleFeePaid = tradingStorage.handleOracleFee(oracleFeeOffered);

-         emit OracleFeeCharged(sender, t.pairIndex, oracleFeePaid);
+         emit OracleFeeCharged(sender, t.pairIndex, oracleFeeOffered);
```

[L-04] SetDev() fails to collect fees for the old `dev` address

Description

The `DomfiRegistry::setDev()` allows the owner to change the `dev` address due to potential requirements:

```
    function setDev(address newDev) public onlyOwner {
        if (newDev == address(0)) revert NullAddr();
        if (newDev == gov || newDev == manager || newDev == owner()) {
            revert HasAlreadyRole(newDev);
        }

        address oldDev = dev;
        if (newDev == oldDev) {
            revert IdenticalDevAddress();
        }

        dev = newDev;
        emit DevUpdated(oldDev, newDev);
    }
```

However, this function fails to ensure that the `DomfiTradingStorage::devFees` and `DomfiTradingStorage::oracleFees` are collected before changing the `dev` address.

It is recommended that the `setDev()` call the `DomfiTradingStorage::claimOracleFees()` and `DomfiTradingStorage::claimFees()` functions before changing the address.

[L-05] ReceiveAssets() missing onlyCallbacks allows public vault donation

Description

The `IDomfiVault` interface comment suggests that only callbacks should be allowed to call the `receiveAssets()` function:

```
    // onlyCallbacks
    function sendAssets(uint256 assets, address receiver) external;
    function receiveAssets(uint256 assets, address user) external;
```

However, the implementation of `receiveAssets()` allows anyone to call it, essentially donating to the vault's P&L accounting:

```
    function receiveAssets(uint256 assets, address user) external {
        address sender = _msgSender();
        SafeERC20.safeTransferFrom(_assetIERC20(), sender, address(this), assets);
```

It is recommended to add the `onlyCallbacks` modifier to make it in sync with the documentation.

[L-06] Override `_transferOwnership()` in `DomfiRegistry` to avoid role overlap

Description

`setGov()`, `setDev()`, and `setManager()` ensure that an address can hold only one role, including the owner role, as shown below:

```
if (newDev == gov || newDev == manager || newDev == owner()) {
    revert HasAlreadyRole(newDev);
}
```

However, `_transferOwnership()` is not overridden. This allows an address to be assigned as `gov` first and later to become the `owner`, violating the invariant that an address can hold only one role.

Recommendation: Override `_transferOwnership()` and add checks to ensure that the new owner does not already hold any role.

[L-07] setValue() and setValues() use `<` instead of `<=` to validate the timestamp

Description

`setValue()` and `setValues()` are used to set the price for a specific `productId`. According to the NatSpec:

```
* @dev Reverts if any timestamp is <= previous timestamp
```

However, the implementation reverts only when `timestamp < feeds[productId].timestamp`, not when it is equal. This contradicts the NatSpec and allows updates with the same timestamp.

Recommendation: Use `<=` instead of `<` when validating the timestamp.

[L-08] FeeOk does not check whether `name != bytes(0)`

Description

`addFee()` is used to add new fees, and uses the `_feeOk()` modifiers to verify their configuration. The problem is that these checks do not verify that `name != bytes(0)`, while `_feeListed()` determines whether a pair exists by checking if `name == bytes(0)`.

This means a pair or fee could be added with `name == bytes(0)`, and later `_feeListed()` would incorrectly treat it as non-existent even though it was actually added. This leads to an inconsistent state and unexpected behavior.

Recommendation: Add a validation in `_feeOk()` to check that `name != bytes(0)` and revert otherwise.

[L-09] ScaleVariables() division by zero prevents full withdrawal

Description

In `withdraw()` and `redeem()`, code calls `scaleVariables()` to scale the `accPnlPerToken` variable based on the new supply:

```
  function redeem(uint256 shares, address receiver, address owner)
      public
      override
      checks(shares)
      returns (uint256)
  {
      require(shares <= maxRedeem(owner), "ERC4626: redeem more than max");

      withdrawRequests[owner][currentEpoch] -= shares;
```

```
        uint256 assets = previewRedeem(shares);
        scaleVariables(shares, false);

        _withdraw(_msgSender(), receiver, owner, assets, shares);
        return assets;
    }

    function scaleVariables(uint256 shares, bool isDeposit) private {
        uint256 supply = totalSupply();

        if (accPnlPerToken < 0) {
            // slither-disable-next-line divide-before-multiply
            accPnlPerToken = accPnlPerToken * supply.toInt256()
                / (isDeposit ? (supply + shares).toInt256() : (supply - shares).toInt256());
        }
    }
```

The issue is that in `scaleVariables()` code divides by `supply - shares`, which can be 0 if the last user withdraws all of their shares, and the transaction would revert because of division by zero. Code should check for 0 when dividing by `supply - shares` and update `accPnlPerToken` to a correct value without performing division.

[L-10] Remove `removeCollateral` and `topUpCollateral` may reduce position below limit

Description

When the user calls `removeCollateral` and `topUpCollateral`, the code changes the positions of collateral and leverage, and because of rounding errors, the position size will be changed:

```
        uint256 tradeSize = t.collateral.mulDiv(t.leverage, 100, Math.Rounding.Ceil);
        uint256 newCollateral = t.collateral - removeAmount;

        uint256 leverageNumerator = tradeSize * PRECISION_6;
        uint256 leverageDenominator = newCollateral * 1e4;

        uint32 newLeverage = (leverageNumerator / leverageDenominator).toUint32();

        // check for precision loss in the `newLeverage` calculation
        uint256 remainder = leverageNumerator % leverageDenominator;

        if (remainder != 0) {
            newCollateral = tradeSize * 1e2 / newLeverage;

            if (newCollateral < t.collateral) {
                removeAmount = t.collateral - newCollateral;
            } else {
                revert WrongParams();
            }
        }
```

The issue is that the new position size can be lower than the old position size, and it can be slightly lower than `pairMinLevPos`. As the position size changes, the code should check that the new position size is bigger than `pairMinLevPos` too.

[L-11] OpenTradeMarketCallback() uses prefee collateral for price impact calculation

Description

In function `openTradeMarketCallback()`, the first code calls `getTradePriceImpact()` with the raw collateral amount and calculates the price after impact:

```
        (uint256 priceImpactP, uint256 priceAfterImpact) =
TradingCallbacksLib.getTradePriceImpact(
            a.price,
            trade.collateral,
            trade.leverage,
            tradingStorage,
            trade.pairIndex,
            true,
            trade.buy
        );
```

And then in the `registerTrade()` code, it deducts the fee from the position collateral:

```
        // 2.1 Charge opening fee
        {
            (uint256 reward, uint256 vaultReward) = tradingStorage.handleOpeningFees(
                trade.pairIndex,
                latestPrice,
                trade.collateral.mulDiv(trade.leverage, 100, Math.Rounding.Ceil),
                trade.leverage,
                trade.buy
            );

            trade.collateral -= reward;

            emit DevFeeCharged(tradeId, trade.trader, reward);

            if (vaultReward > 0) {
                IDomfiVault vault = IDomfiVault(registry.getContractAddress("vault"));
                tradingStorage.transferUsdc(address(tradingStorage), address(this),
vaultReward);
                vault.distributeReward(vaultReward);
                trade.collateral -= vaultReward;
                emit VaultOpeningFeeCharged(tradeId, trade.trader, vaultReward);
            }
     }
```

As a result, the price after impact will be calculated based on the position size bigger than the real opened position.

[L-12] Tp/sl validation ignores pending remove-collateral operation

Description

The `updateTp` and `updateSl` functions validate take-profit and stop-loss values against the current leverage, but do not check for pending `removeCollateral` operations that will increase leverage when executed.

```
uint256 maxTpDist = (t.openPrice * MAX_GAIN_P) / (initialLeverage > t.leverage ?
initialLeverage : t.leverage);
// ... snip ...
uint256 maxSlDist = (t.openPrice * maxSlP) / t.leverage;
```

Since removing collateral increases leverage and reduces the allowed TP/SL distance, users can set TP/SL values that pass validation at the current leverage but become out of bounds after the pending operation completes, and will be updated using `correctTp` and `correctToNullSl`.

Recommendations:

Consider modifying `updateTp` and `updateSl` to also check for pending `removeCollateral` triggers.

[L-13] Callbacks can execute on stale state after trigger timeout

Description

Automation callbacks (`executeAutomationOrderCallback` , `handleRemoveCollateral` , etc.) do not validate whether their associated triggers have timed out before executing. While the timeout mechanism allows users to regain control after `triggerTimeout` blocks by expiring the trigger lock, callbacks can still execute on stale state if they run after the timeout but before the pending order is cleaned up by the upKeep.

Recommendations:

Consider adding timeout validation in all callbacks.

[L-14] Open trade does not validate collateral sufficiency for opening fee

Description

The `openTrade` function does not validate whether the provided collateral is sufficient to cover the opening fees before accepting the trade request. In `openTrade` , the protocol accepts the user's collateral without any validation against opening fees:

```
tradingStorage.transferUsdc(sender, address(tradingStorage), t.collateral);
```

The opening fees are only calculated and deducted later in the `registerTrade` callback:

```
// 2.1 Charge opening fee
(uint256 reward, uint256 vaultReward) = tradingStorage.handleOpeningFees(
    trade.pairIndex,
    latestPrice,
    trade.collateral.mulDiv(trade.leverage, 100, Math.Rounding.Ceil),
    trade.leverage,
    trade.buy
);

trade.collateral -= reward; // Will underflow if reward > collateral

// ...

if (vaultReward > 0) {
    // ...
    trade.collateral -= vaultReward; // Will also underflow if total fees > collateral
}
```

This allows users to submit trades that will inevitably fail in the callback due to underflow, causing them to waste their oracle fees on doomed transactions.

Recommendations:

Consider adding upfront validation in `openTrade` to check if the collateral can cover the "maximum" opening fees.

[L-15] Pause check in `executeAutomationOrder` reverts instead of returning status

Description

The `executeAutomationOrder` function is called by `TradesUpKeep` to process automation orders in batches. When the protocol is paused, opening orders should be blocked, but the function reverts instead of returning a status:

```
if (orderType == IDomfiTradingStorage.LimitOrder.OPEN) {
    if (!tradingStorage.hasOpenLimitOrder(trader, pairIndex, index)) {
        return (IDomfiTrading.AutomationOrderStatus.NO_LIMIT, orderId);
    }
    isNotPaused();  // Reverts entire transaction
}
```

This is inconsistent with other checks in the same function that return status codes (`NO_LIMIT`, `NO_TRADE`, `NO_SL`, `NO_TP`, `PENDING_TRIGGER`).

When TradesUpKeep processes multiple orders in a single transaction, if it encounters a paused OPEN order, the entire batch reverts.

This blocks processing of subsequent orders (including close orders like SL, TP, or liquidations) that should execute even when the protocol is paused.

Recommendations:

Consider returning a status code instead of reverting to allow batch processing to continue.

[L-16] Redundant `cancelReason` condition in `closeTradeMarketCallback`

Description

The `closeTradeMarketCallback` sets `cancelReason` to either `NO_TRADE` or `NONE`:

```
CancelReason cancelReason = t.leverage == 0 ? CancelReason.NO_TRADE : CancelReason.NONE;
```

Then checks:

- `cancelReason != NO_TRADE` and
- `cancelReason == NONE`

Given the assignment, `cancelReason == NONE` already implies `cancelReason != NO_TRADE`, so the first condition is redundant.

Recommendations:

Consider replacing the condition with a single check:

```
if (cancelReason == CancelReason.NONE) { ... }
```

[L-17] Remove `removeCollateral` does not validate `newLeverage`

Description

In `DomfiTrading.removeCollateral`, the contract recomputes the position leverage after decreasing collateral:

```
uint256 tradeSize = t.collateral.mulDiv(t.leverage, 100, Math.Rounding.Ceil);
uint256 newCollateral = t.collateral - removeAmount;

uint256 leverageNumerator = tradeSize * PRECISION_6;
uint256 leverageDenominator = newCollateral * 1e4;

uint32 newLeverage = (leverageNumerator / leverageDenominator).toUint32();
```

However, `newLeverage` is never validated against the pair's leverage constraints (e.g. `pairMaxLeverage`) before:

1. Charging the oracle/automation fee; and
2. Enqueuing a `REMOVE_COLLATERAL` automation order.

The downstream callback logic will eventually reject the operation if `newLeverage` violates the leverage bounds, but by that time, the user has already paid the oracle fee for a request that will be rejected.

Recommendations:

Consider validating `newLeverage` in `removeCollateral` before charging oracle fees or storing the pending order.

[L-18] DomfiVault::maxRedeem can underflow violating ERC4626 expectations

Description

`DomfiVault::maxRedeem` caps the redeemable shares with `totalSupply() - 1`:

```
function maxRedeem(address owner) public view override returns (uint256) {
    return IDomfiOpenPnl(registry.getContractAddress("openPnl")).nextEpochValuesRequestCount()
        == 0 ? Math.min(withdrawRequests[owner][currentEpoch], totalSupply() - 1) : 0;
}
```

If `totalSupply() == 0`, this expression reverts via underflow before `Math.min` can clamp the result, instead of returning `0` as expected by the ERC-4626 view function.

This behavior is inconsistent with the ERC-4626 specification's expectation that view functions are pure introspection helpers.

Recommendations:

Consider handling the zero-supply case explicitly.

[L-19] Inconsistent rounding of `accPnlPerToken` in `updateAccPnlPerTokenUsed`

Description

The protocol applies asymmetric rounding in favor of the vault when updating `accPnlPerToken` during trade settlement. When traders make a profit, `DomfiVault::sendAssets` is invoked and `accPnlDelta` is rounded **up** using `Math.Rounding.Ceil`, ensuring the vault accounts for at least the full loss. Conversely, when traders incur a loss, `DomfiVault::receiveAssets` is used and `accPnlDelta` is implicitly rounded **down**, again favoring the vault.

However, this rounding strategy is not applied consistently in `updateAccPnlPerTokenUsed`. In this function, `accPnlPerToken` is updated using:

```
accPnlPerToken += delta * int32(PRECISION_6) / supply.toInt256();
```

This calculation always rounds toward zero due to integer division, regardless of whether `delta` represents profit or loss. As a result, rounding always biases the result downward in absolute terms. In epochs where the net PnL across all positions is positive (i.e., traders are in profit), the increase to `accPnlPerToken` is rounded down, reducing the accounted loss for the vault. This behavior is inconsistent with the rounding logic used in `sendAssets` and `receiveAssets`, and favors the vault users against the protocol.

It is recommended to apply directional rounding consistent with the rest of the system, round up when delta > 0 (traders profit, vault loss) and round down when delta < 0 (traders' loss, vault profit).

[L-20] `PairInfos::getTradeLiquidationPrice` does not return correct liquidation price

Description

`pairInfos::getTradeLiquidationPrice` returns the liquidation price at which a trade is supposed to be liquidated. However, it does not return the correct liquidation price. `getTradeLiquidationPrice` calls `getTradeLiquidationPricePure` to get the liquidation price. In `getTradeLiquidationPricePure`, when calculating `liqDistance`, it is rounded down:

```
int256 liqPriceDistance = (openPrice.toInt256() * targetCollateralAfterFees)
            / signedCollateral * int8(PRECISION_2) / int32(leverage);
```

Due to this rounding loss, liquidation does not occur at the price returned by `getTradeLiquidationPrice`. As demonstrated in the PoC, liquidation only triggers after the price moves further (e.g., by an additional 1e8) to compensate for the truncated amount. It is recommended to round up the `liqPriceDistance` to mitigate this issue.

[L-21] Function `removePendingRemoveCollateral` would always revert

Description

Function `removePendingRemoveCollateral()` is supposed to remove the pending collateral removal requests and their triggers related to that request:

```
function removePendingRemoveCollateral(uint256 orderId)
        external
        onlyTradesUpKeep
        notDone
        returns (IDomfiTrading.AutomationOrderStatus)
    {
        IDomfiTradingStorage tradingStorage =
            IDomfiTradingStorage(registry.getContractAddress("tradingStorage"));
        IDomfiTradingStorage.PendingRemoveCollateral memory pending =
            tradingStorage.getPendingRemoveCollateral(orderId);

        if (pending.trader == address(0)) {
            return IDomfiTrading.AutomationOrderStatus.NO_TRADE;
        }

        if (!checkNoPendingTriggers(pending.trader, pending.pairIndex, pending.index)) {
            return IDomfiTrading.AutomationOrderStatus.PENDING_TRIGGER;
        }

        tradingStorage.unregisterTrigger(
            pending.trader,
            pending.pairIndex,
            pending.index,
            IDomfiTradingStorage.LimitOrder.REMOVE_COLLATERAL
        );
        tradingStorage.unregisterPendingRemoveCollateral(orderId);

        return IDomfiTrading.AutomationOrderStatus.SUCCESS;
    }
```

There is no trigger check, and if there is a trigger registered for that position, then the code would revert. The issue is that if there is a pending remove collateral request, then there would always be a trigger too, and checking for no trigger will always revert for pending requests, and this function will always revert.

Recommendations

Remove the `!checkNoPendingTriggers()` check and ensure the pending request and the trigger being removed actually exist.

[L-22] Oracle fee cap enables griefing via dust orders draining subsidized gas

Description

These execution costs significantly exceed the maximum oracle fee that can be charged to the trader. Additionally, there is currently no minimum order size enforcement. As a result, an attacker can repeatedly create very small (dust) orders, paying the low oracle fee, while forcing the protocol to continuously subsidize expensive upkeep executions.

This allows an attacker to economically grief the protocol by draining the ETH allocated for oracle and automation operations, potentially degrading service for legitimate traders.

Recommendations

Describe your recommendation here.

Increase the `MAX_ORACLE_FEE` or add a minimum order size check.

[L-23] Deposit/withdraw should not change share prices

Description

In ERC4626, deposit and withdraw operations should not change the share price; otherwise, if one user deposits or withdraws tokens, it would affect the other users. Right now, in the current implementation, the share price is calculated based on `accPnlPerToken`, which is also responsible for keeping the accumulated PnL. This creates a contradicted situation:

1. During deposit/withdraw, the `accPnlPerToken` should be scaled to reflect the correct accumulated PnL.
2. During deposit/withdraw, the share price should not be changed, and so `accPnlPerToken` should not be changed.

The issue is that in the current code, instead of tracking the absolute value of accumulated PnL, the code keeps track of accumulated PnL per share token and uses it for both purposes mentioned above. In the current code, this does not lead to a bug because:

1. When `accPnlPerToken > 0`, code does not scale its value inside function `scaleVariables()`, which is another bug, and if it is fixed then it would result in a share price change during deposit withdraw.
2. When `accPnlPerToken < 0`, code scales its value, but because of a design choice, when `accPnlPerToken < 0`, then code does not consider `accPnlPerToken` value for share price calculation.

So as you can see, there is a bug here, but because of another bug and a design choice, this bug's effect is canceled. However, if the other bug is fixed or the design choice is changed, then the bug will show itself.

Recommendations

Keep track of the accumulated PnL instead of the `accPnlPerToken` and perform the division by total shares inside the `updateShareToAssetsPrice()` function.

[L-24] Deposit and mint functions may validate against outdated max supply

Description

The `OstiumVault::tryUpdateCurrentMaxSupply()` can be called publicly and is persistently called inside `sendAssets()`, `receiveAssets()`, and `updateAccPnlPerTokenUsed()` to ensure a mint cap is maintained:

```
  function tryUpdateCurrentMaxSupply() public {
      if (block.timestamp - lastMaxSupplyUpdateTs >= 24 hours) {
          currentMaxSupply =
                totalSupply() * (uint16(100) * PRECISION_2 + maxSupplyIncreaseDailyP) /
(PRECISION_2 * uint16(100));
            lastMaxSupplyUpdateTs = uint32(block.timestamp);

            emit CurrentMaxSupplyUpdated(currentMaxSupply);
        }
    }
```

However, the `deposit()`, `mint()`, and `mintWithDiscountAndLock()` fail to update the supply cap, which can lead to a higher max supply than intended or revert a mint/deposit transaction due to stale maximum current supply.

It is recommended to add `tryUpdateCurrentMaxSupply()` to the calls above to update the max supply before checking against the `maxMint()`.

[L-25] Allowance bypass in `makeWithdrawRequest` and `cancelWithdrawRequest`

Description

The `OstiumVault::makeWithdrawRequest()` and `OstiumVault::cancelWithdrawRequest()` allow users and addresses with allowances for a particular user to create or cancel a withdrawal request.

The allowance is checked for the given shares:

```
        if (sender != owner && (allowance == 0 || allowance < shares)) {
            revert NotAllowed(sender);
        }
```

However, this allowance is not reduced or accounted for, which allows the address with allowance to actually create or cancel withdrawal requests by calling the function multiple times.

This scenario can be replicated as follows:

1. A user provides allowance (X) smaller than the actual share holdings (Y) to an address `A` (which can probably be an external integration or a strategy-based contract).
2. Even though address `A` has a lesser allowance than the share holdings, it can spam `makeWithdrawRequest()` and `cancelWithdrawRequest()` till it inflates or deflates `totalSharesBeingWithdrawn()` respectively.

This opens up a griefing attack vector that can be leveraged to time grief withdrawals by canceling at the end of every epoch or simply grief `transfer()` / `transferFrom()` functions that utilize `totalSharesBeingWithdrawn()`, potentially blocking transfers unless explicit allowance is set back to 0.

It is recommended to separate actual share allowance concerns from the create/cancel withdrawal allowances; otherwise, document this behavior.

[L-26] Remove `removePair()` can be DOSed by a malicious user

Description

`removePair()` is used to remove a pair so it cannot be used to open a trade.

```
function removePair(uint16 _pairIndex) external onlyGov pairListed(_pairIndex) {
    if
(IOstiumTradingStorage(registry.getContractAddress('tradingStorage')).pairTradersCount(_pairIndex)
> 0) {
        revert PairNotEmpty();
    }

    Pair memory p = pairs[_pairIndex];

    isPairListed[p.from][p.to] = false;
    isPairIndexListed[_pairIndex] = false;

    emit PairRemoved(_pairIndex, p.from, p.to);
}
```

Before removing a pair, the function checks that there are no active trades. A malicious user could open a trade with very high collateral and an extreme take profit solely to block the pair from being removed, effectively creating DoS.

Recommendation: Implement a `forceRemovePair()` function that removes the pair without checking for active trades. This avoids the DoS risk while still allowing governance to unlist pairs when necessary.

[L-27] `unlockDeposit()` should check `getPnlPerToken() > maxAccPnlPerToken()`

The check `getPnlPerToken() > maxAccPnlPerToken()` is to make sure pool has enough funds to cover the discount, the issue is that code performs the check then decrease the value of the `accRewardsPerToken` but it should be the other way around. Update value of `accRewardsPerToken` before the `getPnlPerToken() > maxAccPnlPerToken()` check to fix the issue.

```
        if (getPnlPerToken() > maxAccPnlPerToken().toInt256()) {
            revert NotEnoughAssets();
        }

        accRewardsPerToken -= uint256(discount);
```

[L-28] "Check if SL/TP is reached" is repeated two times

In function `getAutomationOpenOrderCancelReason()`, code performs SL/TP check two times. To fix this, remove one of these exact code snippets.

```
        // Check if TP is reached
        if (limit.tp > 0  && (limit.buy ? priceAfterImpact >= limit.tp : priceAfterImpact <=
limit.tp) ) {
            return IDomfiTradingCallbacks.CancelReason.TP_REACHED;
        }
        // Check if SL is reached
        if ( limit.sl > 0  && (limit.buy ? priceAfterImpact <= limit.sl : priceAfterImpact >=
limit.sl) ) {
            return IDomfiTradingCallbacks.CancelReason.SL_REACHED;
        }
```

[L-29] `startNewEpoch()` blends `spotPnl` even when median fallback is triggered

In function `startNewEpoch()` when the recorded pnl count are less than `requestsCount`, code doesn't use the median value and fall back to the `currentEpochPositiveOpenPnl` but later code blend with `spotPnl` regardless of the `requestsCount`:

```
        // If all responses arrived, use mean, otherwise it means we forced a new epoch,
        // so as a safety we use the last epoch value
        int256 newEpochOpenPnl = _nextEpochValues.length >= requestsCount
            ? medianEpochValue
            : currentEpochPositiveOpenPnl;

        // Note: final newEpochOpenPnl is a blend of the current spot price and median pnl
        int256 spotPnl = getOpenPnl();
        newEpochOpenPnl = (newEpochOpenPnl * 6 + spotPnl * 4) / 10;
```

The issue is that when there isn't enough records code just use `(currentEpochPositiveOpenPnl * 6 + spotPnl * 4) / 10` for the new epoch pnl, and this bypass the safety mechanism (falling back to `currentEpochPositiveOpenPnl` when there isn't enough pnl records). This may be intentional behavior.