Gains Network Security Review

Pashov Audit Group

Conducted by: Said, astros, pontifex, gandu, sashik-eth

May 26th 2025 - June 6th 2025

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Introduction 3
4. About Gains Network 3
5. Risk Classification 4
   5.1. Impact 4
   5.2. Likelihood 4
   5.3. Action required for severity levels 4
6. Security Assessment Summary 5
7. Executive Summary 6
8. Findings 9
   8.1. Critical Findings 9
      [C-01] Decrease position can be abused to withdraw PnL twice 9
   8.2. High Findings 14
      [H-01] Holding fees accounting discrepancy when collateral is less than fees 14
      [H-02] GetTradeSkewPriceImpactP function ignores trade contract version 15
      [H-03] Decrease position's updateTradeSuccess will use wrong availableCollateralInDiamond 16
      [H-04] Traders could lose assets when increasing counter trade position size 17
   8.3. Medium Findings 22
      [M-01] Decreasing leverage with negative PnL may misdirect assets to the vault 22
      [M-02] Difference between charged and paid fees 25
      [M-03] Manual negative PnL realization can incorrectly realize losses 27
      [M-04] Wrong rounding direction when calculating leverage delta for counter trades 28
      [M-05] Open price for counter trades with excess collateral calculated wrongly 29
      [M-06] Incorrect LINK fee is charged for the PNL_WITHDRAWAL order type 31
   8.4. Low Findings 34
      [L-01] SetRoles prevents self-assignment for GOV_TIMELOCK only 34

[L-02] State update in executeManualHoldingFeesRealizationCallback may cause fee errors 34
[L-03] Admin can change price impact parameters for active trades 36
[L-04] Rounding errors can favor users by a better liquidation price 36
[L-05] updateFeeTierPoints is not properly called during several operations 38
[L-06] MillisecondsPerBlock value for Ape chain is outdated 42
[L-07] Borrowing fees not accrued before updating fee per block cap parameters 43
[L-08] Less burn amount if negative PnL is realized early 44
[L-09] Position size validation uses pre-fee amounts 45
[L-10] ExecuteManualNegativePnlRealizationCallback could revert on 0 transfer 46
[L-11] Reverted request IDs stay pending for the protocol 48
[L-12] Rounding errors causes minimal position size less than expected 48

1. About Pashov Audit Group

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

2. Disclaimer

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

3. Introduction

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

4. About Gains Network

Gains Network is a liquidity-efficient decentralized leveraged trading platform. Trades are opened with DAI, USDC or WETH collateral, regardless of the trading pair. The leverage is synthetic and backed by the respective gToken vault, and the GNS token. Trader profit is taken from the vaults to pay the traders PnL (if positive), or receives trader losses from trades if their PnL was negative.

5. Risk Classification

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

5.1. Impact

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

5.2. Likelihood

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

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - c9bc0b058b0bd336ff99ce31d50d152b32a32d43

fixes review commit hash - 6af8860727d71f5f44c815c1f34ca0f64480ce54

Scope

The following smart contracts were in scope of the audit:

- `GToken`
- `GNSAddressStore`
- `GNSDiamondCut`
- `GNSDiamondStorage`
- `GNSBorrowingFees`
- `GNSFundingFees`
- `GNSPairsStorage`
- `GNSPriceAggregator`
- `GNSPriceImpact`
- `GNSTradingCallbacks`
- `GNSTradingInteractions`
- `GNSTradingStorage`
- `interfaces/`
- `BorrowingFeesUtils`
- `ConstantsUtils`
- `FundingFeesUtils`
- `PackingUtils`
- `PairsStorageUtils`
- `PriceAggregatorUtils`
- `PriceImpactUtils`
- `StorageUtils`
- `TradeManagementCallbacksUtils`
- `TradingCallbacksUtils`
- `TradingCommonUtils`
- `TradingInteractionsUtils`
- `TradingStorageGetters`
- `UpdateLeverageLifecycles`
- `DecreasePositionSizeUtils`
- `IncreasePositionSizeUtils`
- `UpdatePositionSizeLifecycles`
- `Timelock`

7. Executive Summary

Over the course of the security review, Said, astros, pontifex, gandu, sashik-eth engaged with Gains Network to review Gains Network. In this period of time a total of **23** issues were uncovered.

Protocol Summary

Protocol Name	Gains Network
Repository	https://github.com/GainsNetwork-org/gTrade-contracts
Date	May 26th 2025 - June 6th 2025
Protocol Type	Leveraged trading platform

Findings Count

Severity	**Amount**
Critical	1
High	4
Medium	6
Low	12
Total Findings	**23**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Decrease position can be abused to withdraw PnL twice	Critical	Resolved
[H-01]	Holding fees accounting discrepancy when collateral is less than fees	High	Resolved
[H-02]	GetTradeSkewPriceImpactP function ignores trade contract version	High	Resolved
[H-03]	Decrease position's updateTradeSuccess will use wrong availableCollateralInDiamond	High	Resolved
[H-04]	Traders could lose assets when increasing counter trade position size	High	Resolved
[M-01]	Decreasing leverage with negative PnL may misdirect assets to the vault	Medium	Resolved
[M-02]	Difference between charged and paid fees	Medium	Acknowledged
[M-03]	Manual negative PnL realization can incorrectly realize losses	Medium	Resolved
[M-04]	Wrong rounding direction when calculating leverage delta for counter trades	Medium	Resolved
[M-05]	Open price for counter trades with excess collateral calculated wrongly	Medium	Resolved
[M-06]	Incorrect LINK fee is charged for the PNL_WITHDRAWAL order type	Medium	Acknowledged
[L-01]	SetRoles prevents self-assignment for GOV_TIMELOCK only	Low	Resolved
[L-02]	State update in executeManualHoldingFeesRealizationCallback may cause fee errors	Low	Resolved
[L-03]	Admin can change price impact parameters for active trades	Low	Acknowledged
[L-04]	Rounding errors can favor users by a better liquidation price	Low	Acknowledged

[L-05]	updateFeeTierPoints is not properly called during several operations	Low	Acknowledged
[L-06]	MillisecondsPerBlock value for Ape chain is outdated	Low	Acknowledged
[L-07]	Borrowing fees not accrued before updating fee per block cap parameters	Low	Acknowledged
[L-08]	Less burn amount if negative PnL is realized early	Low	Acknowledged
[L-09]	Position size validation uses pre-fee amounts	Low	Resolved
[L-10]	ExecuteManualNegativePnlRealizationCallback could revert on 0 transfer	Low	Acknowledged
[L-11]	Reverted request IDs stay pending for the protocol	Low	Resolved
[L-12]	Rounding errors causes minimal position size less than expected	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Decrease position can be abused to withdraw PnL twice

Severity

Impact: High

Likelihood: High

Description

When users decrease their position, it will eventually trigger `executeDecreasePositionSizeMarket`.

```
function executeDecreasePositionSizeMarket(
        ITradingStorage.PendingOrder memory _order,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) external {
        // 1. Prepare vars
        ITradingStorage.Trade memory partialTrade = _order.trade;
        ITradingStorage.Trade memory existingTrade = _getMultiCollatDiamond
          ().getTrade(
            partialTrade.user,
            partialTrade.index
        );
        IUpdatePositionSizeUtils.DecreasePositionSizeValues memory values;

        // 2. Refresh trader fee tier cache
        TradingCommonUtils.updateFeeTierPoints(
            existingTrade.collateralIndex,
            existingTrade.user,
            existingTrade.pairIndex,
            0
        );

        // 3. Base validation (trade open, market open)
        ITradingCallbacks.CancelReason cancelReason = _validateBaseFulfillment
          (existingTrade);

        // 4. If passes base validation, validate further
        if (cancelReason == ITradingCallbacks.CancelReason.NONE) {
            // 4.1 Prepare useful values
            //(position size delta, closing fees, borrowing fees, etc.)
>>>         values = DecreasePositionSizeUtils.prepareCallbackValues
  (existingTrade, partialTrade, _answer);

            // 4.2 Further validation
            cancelReason = DecreasePositionSizeUtils.validateCallback
              (existingTrade, _order, values, _answer);

            // 5. If passes further validation, execute callback
            if (cancelReason == ITradingCallbacks.CancelReason.NONE) {
                // 5.1 Send collateral delta
                //(partial trade value - fees) if positive or remove from trade collateral if nega
                // Then update trade collateral / leverage in storage, and reset
                // trade borrowing fees
                DecreasePositionSizeUtils.updateTradeSuccess
                  (existingTrade, values, _answer);

                // 5.2 Distribute closing fees
                TradingCommonUtils.processFees(
                  existingTrade,
                  values.positionSizeCollateralDelta,
                  _order.orderType
                );
            }
        }

        // 6. If didn't pass validation and trade exists, charge gov fee and
        // remove corresponding OI
        if (cancelReason != ITradingCallbacks.CancelReason.NONE)
            DecreasePositionSizeUtils.handleCanceled
              (existingTrade, cancelReason, _answer);

        // 7. Close pending decrease position size order
        _getMultiCollatDiamond().closePendingOrder(_answer.orderId);

        emit IUpdatePositionSizeUtils.PositionSizeDecreaseExecuted(
            _answer.orderId,
            cancelReason,
            existingTrade.collateralIndex,
            existingTrade.user,
            existingTrade.pairIndex,
            existingTrade.index,
            existingTrade.long,
            _answer.current,
            _getMultiCollatDiamond().getCollateralPriceUsd
              (existingTrade.collateralIndex),
            partialTrade.collateralAmount,
            partialTrade.leverage,
```

```
            values
        );
    }
```

Inside `executeDecreasePositionSizeMarket`, it will calculate all values that will be used to update the trade inside `DecreasePositionSizeUtils.prepareCallbackValues`.

```
function prepareCallbackValues(
        ITradingStorage.Trade memory _existingTrade,
        ITradingStorage.Trade memory _partialTrade,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) internal view returns
      (IUpdatePositionSizeUtils.DecreasePositionSizeValues memory values) {
        // 1. Calculate position size delta and existing position size
        bool isLeverageUpdate = _partialTrade.leverage > 0;


                        isLeverageUpdate ? _existingTrade.collateralAmount : _partialTrade.collat
            isLeverageUpdate ? _partialTrade.leverage : _existingTrade.leverage
        );

            _existingTrade.collateralAmount,
            _existingTrade.leverage
        );

        // 2. Calculate partial trade closing fees

            _existingTrade.collateralIndex,
            _existingTrade.user,
            _existingTrade.pairIndex,
            values.positionSizeCollateralDelta,
            _existingTrade.isCounterTrade
        );

        // 3. Calculate new collateral amount and leverage

                values.newCollateralAmount = _existingTrade.collateralAmount - _partialTrade.coll
        values.newLeverage = _existingTrade.leverage - _partialTrade.leverage;

        // 4. Apply spread and price impact to answer.current
        (values.priceImpact, ) = TradingCommonUtils.getTradeClosingPriceImpact(
            ITradingCommonUtils.TradePriceImpactInput(
                _existingTrade,
                _answer.current,
                values.positionSizeCollateralDelta,
                _answer.current
            )
        );

        // 5. Calculate existing trade pnl
>>>     values.existingPnlCollateral =
            (TradingCommonUtils.getPnlPercent(
                _existingTrade.openPrice,
                uint64(values.priceImpact.priceAfterImpact),
                _existingTrade.long,
                _existingTrade.leverage
            ) * int256(uint256(_existingTrade.collateralAmount))) /
            100 /
            int256(ConstantsUtils.P_10);

        // 6. Calculate value sent to trader
>>>     int256 partialTradePnlCollateral =
  (values.existingPnlCollateral * int256(values.positionSizeCollateralDelta)) /
            int256(values.existingPositionSizeCollateral);



                values.availableCollateralInDiamond = _partialTrade.collateralAmount > values.ava
            ? values.availableCollateralInDiamond
            : _partialTrade.collateralAmount;
        // @audit - should this consider pnl withdrawal
>>>     values.collateralSentToTrader = int256(uint256
  (_partialTrade.collateralAmount)) + partialTradePnlCollateral;

        // 7. Calculate existing and new trade liquidation price
        values.existingLiqPrice = TradingCommonUtils.getTradeLiquidationPrice
          (_existingTrade, _answer.current);
        values.newLiqPrice = TradingCommonUtils.getTradeLiquidationPrice(
            _existingTrade,
            _existingTrade.openPrice,
            values.newCollateralAmount,
            values.newLeverage,
            values.closingFeeCollateral +
```

```
                uint256(
                  values.collateralSentToTrader<0?-values.collateralSentToTrader:int256
                )
            _getMultiCollatDiamond().getTradeLiquidationParams
              (_existingTrade.user, _existingTrade.index),
            _answer.current
        );
    }
```

When calculating `existingPnlCollateral`, it uses `_existingTrade.collateralAmount`. Then, when calculating `partialTradePnlCollateral`, it uses a portion of `existingPnlCollateral` based on `values.positionSizeCollateralDelta` and `values.existingPositionSizeCollateral`. However, none of these calculations consider withdrawn or realized PnL.

This will cause users to withdraw again PnL after previously already realized.

Recommendations

Consider `realizedPnlCollateral` when calculating `values.existingPnlCollateral`.

8.2. High Findings

[H-01] Holding fees accounting discrepancy when collateral is less than fees

Severity

Impact: High

Likelihood: Medium

Description

When a highly leveraged trade accumulates positive PnL over time, an accounting discrepancy occurs in how holding fees are handled, leading to potential fund loss for the protocol.

The issue arises in the `realizeHoldingFeesOnOpenTrade` function when `holdingFeesCollateral` exceeds `availableCollateralInDiamond`. In this scenario, only the available collateral is sent to the vault, but the full amount is recorded in `realizedTradingFeesCollateral`:

```
function realizeHoldingFeesOnOpenTrade
      (address _trader, uint32 _index, uint64 _currentPairPrice) external {
        ...

        if (holdingFees.totalFeeCollateral > 0) {
            ...

            if (availableCollateralInDiamond > 0) {

                                amountSentToVaultCollateral = holdingFeesCollateral > availableCo
                    ? availableCollateralInDiamond
                    : holdingFeesCollateral;

                TradingCommonUtils.transferFeeToVault(
                  trade.collateralIndex,
                  amountSentToVaultCollateral,
                  trade.user
                );
            }


                        uint128 newRealizedTradingFeesCollateral = tradeFeesData.realizedTradingF
                uint128
                //(holdingFeesCollateral); // @audit Full amount recorded here

                        tradeFeesData.realizedTradingFeesCollateral = newRealizedTradingFeesColla
            ...
    }
```

The discrepancy occurs when:

- A trade has accumulated holding fees exceeding its available collateral.
- Only a portion of these fees are actually transferred to the vault.
- The full amount is recorded as "realized fees".
- The trader later adds more collateral to the position.

This creates a situation where the protocol has a record of collecting more fees than it actually has, leading to incorrect accounting when calculating available collateral:

```
function getTradeAvailableCollateralInDiamond
      (ITradingStorage.Trade memory _trade) external view returns (uint256) {
        ...
        int256 availableCollateralInDiamondRaw = int256(uint256
          (_trade.collateralAmount)) -
            int256(realizedTradingFeesCollateral) -
            int256(manuallyRealizedNegativePnlCollateral);

        return availableCollateralInDiamondRaw > 0 ? uint256
          (availableCollateralInDiamondRaw) : 0;
    }
```

Consider a scenario:

- Trader opens a position with 0.6 ETH collateral at 500x leverage.
- Position becomes profitable, accumulating 0.82 ETH in holding fees.
- Fees are realized, but only 0.6 ETH transferred to vault, while 0.82 ETH recorded as realized.
- Trader adds 0.5 ETH more collateral (total now 1.1 ETH).
- When closing position, available collateral is calculated as: 1.1 ETH - 0.82 ETH = 0.28 ETH.
- Actual available should be: 0.5 ETH (new collateral only, as original 0.6 ETH was already sent to vault).

The vault is effectively losing 0.22 ETH in this scenario.

Recommendations

When closing the trade, checking for unsent fees when there's enough collateral and send any pending fees to the vault.

[H-02] `GetTradeSkewPriceImpactP` function ignores trade contract version

Severity

Impact: Medium

Likelihood: High

Description

The `PriceImpactUtils.getTradeSkewPriceImpactP` does not take into account a trade contract version. This causes applying the same skew price impact to all contract versions. So independent from whether the skew price impact was applied or not when the trade was opened, a half of the price impact will be applied during closing.

```
function getTradeSkewPriceImpactP(
        uint8 _collateralIndex,
        uint16 _pairIndex,
        bool _long,
        uint256 _tradeOpenInterestUsd, // 1e18 USD
        bool _open
    )
        internal
        view
        returns (
            int256 priceImpactP // 1e10 (%)
        )
    {
        IPriceImpact.PriceImpactValues memory v;

        v.tradePositiveSkew = (_long && _open) || (!_long && !_open);
        v.depth = _getStorage().pairSkewDepths[_collateralIndex][_pairIndex];
        v.tradeSkewMultiplier = v.tradePositiveSkew ? int256(1) : int256(-1);
>>      v.priceImpactDivider = int256
//(2); // so depth is on same scale as cumul vol price impact

        return
            _getTradePriceImpactP(
                TradingCommonUtils.getPairOiSkewCollateral
                  (_collateralIndex, _pairIndex) *
                    int256(_getMultiCollatDiamond().getCollateralPriceUsd
                      (_collateralIndex)),
                int256(_tradeOpenInterestUsd) * v.tradeSkewMultiplier,
                v.depth,
                ConstantsUtils.P_10,
                ConstantsUtils.P_10
>>          ) / v.priceImpactDivider;
    }
```

Recommendations

Consider not applying the skew price impact at all when closing trades being opened before the update, i.e. contract version < 10 and applying the full size of the skew price impact when increasing positions for such trades.

[H-03] Decrease position's `updateTradeSuccess` will use wrong `availableCollateralInDiamond`

Severity

Impact: Medium

Likelihood: High

Description

Inside the Decrease Position's `updateTradeSuccess`, the closing fee is deducted from the user. However, it is not considered when providing `_values.availableCollateralInDiamond` to the `handleTradeValueTransfer` operation.

```
function updateTradeSuccess(
        ITradingStorage.Trade memory _existingTrade,
        IUpdatePositionSizeUtils.DecreasePositionSizeValues memory _values,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) internal {
        // 1. Update trade in storage
        _getMultiCollatDiamond().updateTradePosition(
            ITradingStorage.Id(_existingTrade.user, _existingTrade.index),
            _values.newCollateralAmount,
            _values.newLeverage,
            _existingTrade.openPrice, // open price stays the same
            ITradingStorage.PendingOrderType.MARKET_PARTIAL_CLOSE, // don't
            // refresh liquidation params
            _values.existingPnlCollateral > 0,
            _answer.current
        );
        // 2. Realize trading fees and negative pnl (leverage decrease)
>>>     _getMultiCollatDiamond().realizeTradingFeesOnOpenTrade(
            _existingTrade.user,
            _existingTrade.index,
            _values.closingFeeCollateral +
                (_values.collateralSentToTrader < 0 ? uint256
                //(-_values.collateralSentToTrader) : 0), // reduces collateral available in diamc
            _answer.current
        );

        // 3. Handle collateral/pnl transfers
        TradingCommonUtils.handleTradeValueTransfer(
            _existingTrade,
            _values.collateralSentToTrader,
>>>         int256(_values.availableCollateralInDiamond)
        );
    }
```

This will cause `handleTradeValueTransfer` to process an incorrect amount of collateral that needs to be pulled from the vault.

Recommendations

Consider including `closingFeeCollateral` when providing `availableCollateralInDiamond` to `handleTradeValueTransfer`.

```
// ...
        // 3. Handle collateral/pnl transfers
        TradingCommonUtils.handleTradeValueTransfer(
            _existingTrade,
            _values.collateralSentToTrader,
-            int256(_values.availableCollateralInDiamond)
+            int256
+ (_values.availableCollateralInDiamond - _values.closingFeeCollateral)
        );
// ...
```

[H-04] Traders could lose assets when increasing counter trade position size

Severity

Impact: High

Likelihood: Medium

Description

When users request an increase in position size, it will eventually be executed, triggering `executeIncreasePositionSizeMarket`.

```
function executeIncreasePositionSizeMarket(
        ITradingStorage.PendingOrder memory _order,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) external {
        // 1. Prepare vars
        ITradingStorage.Trade memory partialTrade = _order.trade;
        ITradingStorage.Trade memory existingTrade = _getMultiCollatDiamond
          ().getTrade(
            partialTrade.user,
            partialTrade.index
        );
        IUpdatePositionSizeUtils.IncreasePositionSizeValues memory values;
        // 2. Refresh trader fee tier cache
        TradingCommonUtils.updateFeeTierPoints(
            existingTrade.collateralIndex,
            existingTrade.user,
            existingTrade.pairIndex,
            0
        );

        // 3. Base validation (trade open, market open)
        ITradingCallbacks.CancelReason cancelReason = _validateBaseFulfillment
          (existingTrade);

        // 4. If passes base validation, validate further
        if (cancelReason == ITradingCallbacks.CancelReason.NONE) {
            // 4.1 Prepare useful values
            //(position size delta, pnl, fees, new open price, etc.)
>>>         values = IncreasePositionSizeUtils.prepareCallbackValues
  (existingTrade, partialTrade, _answer);

            // 4.2 Further validation
>>>         cancelReason = IncreasePositionSizeUtils.validateCallback(
                existingTrade,
                values,
                _answer,
                partialTrade.openPrice,
                _order.maxSlippageP
            );

            // 5. If passes further validation, execute callback
            if (cancelReason == ITradingCallbacks.CancelReason.NONE) {
                // 5.1 Update trade collateral / leverage / open price in
                // storage, and reset trade borrowing fees
                IncreasePositionSizeUtils.updateTradeSuccess
                  (_answer.orderId, existingTrade, values, _answer);
                // 5.2 Distribute opening fees and store fee tier points for
                // position size delta
                TradingCommonUtils.processFees(
                  existingTrade,
                  values.positionSizeCollateralDelta,
                  _order.orderType
                );
            }
        }

        // 6. If didn't pass validation, charge gov fee
        //(if trade exists) and return partial collateral (if any)
        if (cancelReason != ITradingCallbacks.CancelReason.NONE)
>>>         IncreasePositionSizeUtils.handleCanceled
  (existingTrade, partialTrade, cancelReason, _answer);

        // 7. Close pending increase position size order
        _getMultiCollatDiamond().closePendingOrder(_answer.orderId);

        emit IUpdatePositionSizeUtils.PositionSizeIncreaseExecuted(
            _answer.orderId,
            cancelReason,
            existingTrade.collateralIndex,
            existingTrade.user,
            existingTrade.pairIndex,
            existingTrade.index,
            existingTrade.long,
            _answer.current,
            _getMultiCollatDiamond().getCollateralPriceUsd
              (existingTrade.collateralIndex),
```

```
            partialTrade.collateralAmount,
            partialTrade.leverage,
            values
        );
    }
```

Inside `executeIncreasePositionSizeMarket`, `prepareCallbackValues` is called to calculate all values required for increasing the position size, including `values.counterTradeCollateralToReturn` if the position has `isCounterTrade` set to true. In this case, `_partialTrade.collateralAmount` will also be decreased by `values.counterTradeCollateralToReturn`.

```
function prepareCallbackValues(
        ITradingStorage.Trade memory _existingTrade,
        ITradingStorage.Trade memory _partialTrade,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) internal view returns
      (IUpdatePositionSizeUtils.IncreasePositionSizeValues memory values) {
        // 1.1 Calculate position size delta
        bool isLeverageUpdate = _partialTrade.collateralAmount == 0;


                        isLeverageUpdate ? _existingTrade.collateralAmount : _partialTrade.collat
            _partialTrade.leverage
        );

        // 1.2 Validate counter trade and update position size delta if needed
        if (_existingTrade.isCounterTrade) {
>>>         (
  values.isCounterTradeValidated,
  values.exceedingPositionSizeCollateral
) = TradingCommonUtils
                .validateCounterTrade
                  (_existingTrade, values.positionSizeCollateralDelta);

            if
              (values.isCounterTradeValidated && values.exceedingPositionSizeCollateral > 0) {
                if (isLeverageUpdate) {
                    // For leverage updates, simply reduce leverage delta to
                    // reach 0 skew
                    _partialTrade.leverage -= uint24(

                          (values.exceedingPositionSizeCollateral * 1e3) / _existingTrade.collater
                    );
                } else {
                    // For collateral adds, reduce collateral delta to reach 0
                    // skew
                    values.counterTradeCollateralToReturn =
                        (values.exceedingPositionSizeCollateral * 1e3) /
                        _partialTrade.leverage;
>>>                 _partialTrade.collateralAmount -= uint120
  (values.counterTradeCollateralToReturn);
                }



                                        isLeverageUpdate ? _existingTrade.collateralAmount : _par
                    _partialTrade.leverage
                );
            }
        }

        // ...
    }
```

However, if `validateCallback` return non `CancelReason.NONE` and `handleCanceled` is triggered. it will only send `_partialTrade.collateralAmount` to the user and not considering `values.counterTradeCollateralToReturn`.

```
function handleCanceled(
        ITradingStorage.Trade memory _existingTrade,
        ITradingStorage.Trade memory _partialTrade,
        ITradingCallbacks.CancelReason _cancelReason,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) internal {
        // 1. Charge gov fee on trade (if trade exists)
        if (_cancelReason != ITradingCallbacks.CancelReason.NO_TRADE) {
            // 1.1 Distribute gov fee

                _existingTrade.collateralIndex,
                _existingTrade.user,
                _existingTrade.pairIndex
            );
            uint256 finalGovFeeCollateral = _getMultiCollatDiamond
              ().realizeTradingFeesOnOpenTrade(
                _existingTrade.user,
                _existingTrade.index,
                govFeeCollateral,
                _answer.current
            );
            TradingCommonUtils.distributeExactGovFeeCollateral(
                _existingTrade.collateralIndex,
                _existingTrade.user,
                finalGovFeeCollateral
            );
        }

        // 2. Send back partial collateral to trader
        TradingCommonUtils.transferCollateralTo(
            _existingTrade.collateralIndex,
            _existingTrade.user,
>>>         _partialTrade.collateralAmount
        );
    }
```

This will cause the collateral amount returned to users to be inaccurate, resulting in a loss of funds.

Recommendations

Also consider `values.counterTradeCollateralToReturn` when returning funds inside `handleCanceled`

```
//...
        // 2. Send back partial collateral to trader
        TradingCommonUtils.transferCollateralTo(
            _existingTrade.collateralIndex,
            _existingTrade.user,
-            _partialTrade.collateralAmount
+
+             _partialTrade.collateralAmount + values.counterTradeCollateralToReturn
        );
//...
```

8.3. Medium Findings

[M-01] Decreasing leverage with negative PnL may misdirect assets to the vault

Severity

Impact: High

Likelihood: Low

Description

When a leverage decrease is executed for a trade with negative PnL that causes negative `collateralSentToTrader`, collateral is sent to the vault (because `_collateralSentToTrader < _availableCollateralInDiamond`). This can occur if `partialNetPnlCollateral` is negative.

```
function updateTradeSuccess(
        ITradingStorage.Trade memory _existingTrade,
        ITradingStorage.Trade memory _partialTrade,
        IUpdatePositionSizeUtils.DecreasePositionSizeValues memory _values,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) internal {
        // 1. Update trade in storage (realizes pending holding fees)
        _getMultiCollatDiamond().updateTradePosition(
            ITradingStorage.Id(_existingTrade.user, _existingTrade.index),
            _values.newCollateralAmount,
            _values.newLeverage,
            _existingTrade.openPrice, // open price stays the same
            ITradingStorage.PendingOrderType.MARKET_PARTIAL_CLOSE, // don't
            // refresh liquidation params
            _values.priceImpact.positionSizeToken,
            _values.existingPnlPercent > 0,
            _answer.current
        );

        // 2. Calculate remaining success callback values
        prepareSuccessCallbackValues(_existingTrade, _partialTrade, _values);

        // 3. Handle collateral/pnl transfers with vault/diamond/user
>>>     TradingCommonUtils.handleTradeValueTransfer(
            _existingTrade,
            _values.collateralSentToTrader,
            int256(_values.availableCollateralInDiamond)
        );

        if (_values.isLeverageUpdate) {

            // First we realize the partial pnl amount so prev raw pnl = new raw
            // pnl, and then we remove what we sent from trader
            _getMultiCollatDiamond().realizePnlOnOpenTrade(
                _existingTrade.user,
                _existingTrade.index,
                _values.pnlToRealizeCollateral
            );

            // 4.1.2 Realize trading fee (if leverage decrease)
            // Not needed for collateral decreases because closing fee is
            // removed from available collateral in diamond and trade value
            // So vault sends as much as if there was no closing fee, but we
            // send less to the trader, so we always have the partial closing fee in diamond
            // This is especially important for lev decreases when nothing is
            // available in diamond, this will transfer the closing fee from vault
>>>         _getMultiCollatDiamond().realizeTradingFeesOnOpenTrade(
                _existingTrade.user,
                _existingTrade.index,
                _values.closingFeeCollateral,
                _answer.current
            );

            // 4.1.3 Decrease collat available in diamond
            //(if we've sent negative PnL to vault)
            if (_values.collateralSentToTrader < 0) {
>>>             _getMultiCollatDiamond().storeAlreadyTransferredNegativePnl(
                    _existingTrade.user,
                    _existingTrade.index,
                    uint256(-_values.collateralSentToTrader)
                );
            }
        } else {
            // ...
    }
```

Then, since this is a leverage update, `realizeTradingFeesOnOpenTrade` will be triggered. If the trade becomes liquidatable due to the fee, it will be liquidated or else we update the user's `realizedTradingFeesCollateral`.

```
function realizeTradingFeesOnOpenTrade(
        address _trader,
        uint32 _index,
        uint256 _feesCollateral,
        uint64 _currentPairPrice
    ) external returns (uint256 finalFeesCollateral) {
        IFundingFees.FundingFeesStorage storage s = _getStorage();
        IFundingFees.RealizeTradingFeesValues memory v;

        v.trade = TradingCommonUtils.getTrade(_trader, _index);
        v.liqPrice = TradingCommonUtils.getTradeLiquidationPrice
          (v.trade, int256(_feesCollateral), _currentPairPrice);


                IFundingFees.TradeFeesData storage tradeFeesData = s.tradeFeesData[_trader][_inde

                v.newRealizedFeesCollateral = tradeFeesData.realizedTradingFeesCollateral;
        v.newRealizedPnlCollateral = tradeFeesData.realizedPnlCollateral;

        // @audit - this should not be possible on openTrade
        // 1. Check if trade can be liquidated after charging trading fee
        if ((v.trade.long && _currentPairPrice <= v.liqPrice) ||
          (!v.trade.long && _currentPairPrice >= v.liqPrice)) {
>>>         TradingCallbacksUtils._unregisterTrade(
                v.trade,
                0,
                ITradingStorage.PendingOrderType.LIQ_CLOSE,
                _currentPairPrice,
                v.liqPrice,
                _currentPairPrice
            );

            uint256 collateralPriceUsd = _getMultiCollatDiamond
              ().getCollateralPriceUsd(v.trade.collateralIndex);

            emit ITradingCallbacksUtils.LimitExecuted(
                ITradingStorage.Id(address(0), 0),
                v.trade.user,
                v.trade.index,
                0,
                v.trade,
                address(0),
                ITradingStorage.PendingOrderType.LIQ_CLOSE,
                _currentPairPrice,
                _currentPairPrice,
                v.liqPrice,
                ITradingCommonUtils.TradePriceImpact(0, 0, 0, 0, 0, 0),
                -100 * 1e10,
                0,
                collateralPriceUsd,
                false
            );
        } else {
            finalFeesCollateral = _feesCollateral;
            // 2. Send fee delta from vault if total realized fees are above
            // trade collateral, so collateral in diamond always >= 0

                _trader,
                _index,
                v.trade.collateralAmount
            );

            if (finalFeesCollateral > availableCollateralInDiamond) {

                                v.amountSentFromVaultCollateral = finalFeesCollateral - available
                TradingCommonUtils.receiveCollateralFromVault
                  (v.trade.collateralIndex, v.amountSentFromVaultCollateral);

                v.newRealizedPnlCollateral -= int128(int256
                  (v.amountSentFromVaultCollateral));

                                tradeFeesData.realizedPnlCollateral = v.newRealizedPnlCollateral;
            }

            // 3. Register new realized fees
            v.newRealizedFeesCollateral += uint128
```

```
            (finalFeesCollateral - v.amountSentFromVaultCollateral);

                    tradeFeesData.realizedTradingFeesCollateral = v.newRealizedFeesCollateral

            /// @dev This makes the transaction revert in case new available
            // collateral in diamond is < 0
            TradingCommonUtils.getTradeAvailableCollateralInDiamond
              (_trader, _index, v.trade.collateralAmount);

            // 4. Update UI realized pnl mapping

        }

        emit IFundingFeesUtils.TradingFeesRealized(
            v.trade.collateralIndex,
            _trader,
            _index,
            _feesCollateral,
            finalFeesCollateral,
            v.newRealizedFeesCollateral,
            v.newRealizedPnlCollateral,
            v.amountSentFromVaultCollateral
        );
    }
```

In both instances, `getTradeAvailableCollateralInDiamond` is accessed before it is updated. Since `storeAlreadyTransferredNegativePnl` is triggered after `realizeTradingFeesOnOpenTrade`, `getTradeAvailableCollateralInDiamond` returns an incorrect amount when the trade fee is realized.

```
function getTradeAvailableCollateralInDiamondRaw(
        address _trader,
        uint32 _index,
        uint256 _collateralAmount
    ) public view returns (int256) {

        return
            int256(_collateralAmount + uint256
              (tradeFeesData.virtualAvailableCollateralInDiamond)) -
            int256(
                uint256(
                    tradeFeesData.realizedTradingFeesCollateral +
                        tradeFeesData.manuallyRealizedNegativePnlCollateral +
>>>                     tradeFeesData.alreadyTransferredNegativePnlCollateral
                )
            );
    }
```

Recommendations

Trigger `storeAlreadyTransferredNegativePnl` before `realizeTradingFeesOnOpenTrade`.

[M-02] Difference between charged and paid fees

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol charges government fees to cover paid oracle fees:

```
function openTradeMarketCallback
      (ITradingCallbacks.AggregatorAnswer memory _a) external tradingActivated {
<...>
            // Gov fee to pay for oracle cost
            TradingCommonUtils.updateFeeTierPoints
              (t.collateralIndex, t.user, t.pairIndex, 0);


                t.collateralIndex,
                t.user,
                t.pairIndex
            );
```

The problem is that oracle cost depends on the trade position size:

```
function getLinkFee(
        uint8 _collateralIndex,
        address _trader,
        uint16 _pairIndex,
>>      uint256 _positionSizeCollateral, // collateral precision
        bool _isCounterTrade
    ) public view returns (uint256) {
        if (_positionSizeCollateral == 0) return 0;

        (, int256 linkPriceUsd, , , ) = _getStorage
        //().linkUsdPriceFeed.latestRoundData(); // 1e8

        uint256 feeRateMultiplier = _isCounterTrade
            ? _getMultiCollatDiamond().getPairCounterTradeFeeRateMultiplier
              (_pairIndex)
            : 1e3;

        uint256 linkFeeCollateral = _getMultiCollatDiamond
          ().pairOraclePositionSizeFeeP(_pairIndex) *
            feeRateMultiplier *
            TradingCommonUtils.getPositionSizeCollateralBasis(
                _collateralIndex,
                _pairIndex,
>>              _positionSizeCollateral,
                feeRateMultiplier
            ); // avoid stack too deep

        uint256 rawLinkFee = (getUsdNormalizedValue
          (_collateralIndex, linkFeeCollateral) * 1e8) /
            uint256(linkPriceUsd) /
            ConstantsUtils.P_10 /
            100 /
            1e3;

        return _getMultiCollatDiamond().calculateFeeAmount(_trader, rawLinkFee);
    }
```

But in many cases the protocol charges government fees from minimum position size (as in code snippet above) or from decreased position size:

```
function openTradeMarketCallback
      (ITradingCallbacks.AggregatorAnswer memory _a) external tradingActivated {
<...>
        if (v.cancelReason == ITradingCallbacks.CancelReason.NONE) {
            // Return excess counter trade collateral
            if (v.collateralToReturn > 0) {
                TradingCommonUtils.transferCollateralTo
                  (t.collateralIndex, t.user, v.collateralToReturn);
>>              t.collateralAmount = v.newCollateralAmount;
                emit ITradingCallbacksUtils.CounterTradeCollateralReturned
                  (_a.orderId, t.collateralIndex, t.user, v.collateralToReturn);
            }
```

This way the protocol balance can be drained by sending counter trades with unexpected positions causing large LINK fees with adjusting position size during callback.

Recommendations

Consider charging a government fee taking into account the full position size used in `getLinkFee` independent if the order is successful or not.

[M-03] Manual negative PnL realization can incorrectly realize losses

Severity

Impact: Medium

Likelihood: Medium

Description

The `executeManualNegativePnlRealizationCallback` function calculates unrealized PnL using the current price instead of the price after impact. This calculation fails to account for situations where price impact would actually benefit the trader's position.

When skew price impact is large (making `totalPriceImpactP` < 0) and would benefit the position (e.g., making the closing price higher for a long position or lower for a short position), the position might actually be profitable after accounting for price impact. However, since the function calculates PnL using only the current oracle price, it can incorrectly realize a loss when there wouldn't be one after price impact.

```
function executeManualNegativePnlRealizationCallback(
        ITradingStorage.PendingOrder memory _order,
        ITradingCallbacks.AggregatorAnswer memory _a
    ) external {
        if (trade.isOpen) {

                trade,
                _a.current // @audit Uses current price instead of price after
                // impact
            );

                        uint256 unrealizedNegativePnlCollateral = unrealizedPnlCollateral < 0
                ? uint256(-unrealizedPnlCollateral)
                : uint256(0);

            ...

    }
```

Consider a scenario:

- A trader opens a long position on an asset.
- Market conditions lead to a favorable skew price impact (e.g., many shorts, few longs).
- When `executeManualNegativePnlRealizationCallback` is called:
  - Current oracle price shows a small loss (negative PnL).
  - After accounting for skew price impact, the position would actually be profitable.
  - The function incorrectly realizes the negative PnL based on the oracle price.
  - Collateral is sent to the vault unnecessarily.

Recommendations

Check if the price impact favors the position; if it does, use it to calculate the `unrealizedPnlCollateral`.

[M-04] Wrong rounding direction when calculating leverage delta for counter trades

Severity

Impact: Low

Likelihood: High

Description

In `IncreasePositionSizeUtils.prepareCallbackValues`, when a counter trade increases position size by increasing leverage and exceeds the available position size needed to neutralize skew, the function calculates a leverage delta reduction to bring the skew to 0.

However, the calculation `(values.exceedingPositionSizeCollateral * 1e3) / _existingTrade.collateralAmount` uses integer division which rounds down. This

downward rounding means the leverage reduction is insufficient to completely neutralize the skew, leaving a small excess position size that pushes the skew slightly in the opposite direction.

The impact of this rounding depends on the collateral amount. For a single unit of leverage (rounded down), the position size can overshoot by up to `_existingTrade.collateralAmount / 1e3` collateral units. With larger collateral amounts, this can result in significant skew being created in the opposite direction, contradicting the core purpose of counter trades.

```
function prepareCallbackValues(
        ITradingStorage.Trade memory _existingTrade,
        ITradingStorage.Trade memory _partialTrade,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) internal view returns
      (IUpdatePositionSizeUtils.IncreasePositionSizeValues memory values) {
        ...

        // 1.2 Validate counter trade and update position size delta if needed
        if (_existingTrade.isCounterTrade) {
            (
              values.isCounterTradeValidated,
              values.exceedingPositionSizeCollateral
            ) = TradingCommonUtils
                .validateCounterTrade
                  (_existingTrade, values.positionSizeCollateralDelta);

            if
              (values.isCounterTradeValidated && values.exceedingPositionSizeCollateral > 0) {
                if (isLeverageUpdate) {
                    // For leverage updates, simply reduce leverage delta to
                    // reach 0 skew
                    _partialTrade.leverage -= uint24(

                        //(values.exceedingPositionSizeCollateral * 1e3) / _existingTrade.collater
                    );
            ...
    }
```

Recommendations

Modify the calculation to round up instead of down when reducing leverage, ensuring the skew is never pushed in the opposite direction.

[M-05] Open price for counter trades with excess collateral calculated wrongly

Severity

Impact: Medium

Likelihood: Medium

Description

The `openTradeMarketCallback` function opens trades with the `openPrice` (L102) that is calculated based on the price impact that the trade would cause on the market. In case the trade is a `counter` trade some part of the collateral amount (that excess current skew on the pair) would be returned to the user (L106):

```
File: TradingCallbacksUtils.sol
094:     function openTradeMarketCallback
  (ITradingCallbacks.AggregatorAnswer memory _a) external tradingActivated {
095:         ITradingStorage.PendingOrder memory o = _getPendingOrder
  (_a.orderId);
096:
097:         if (!o.isOpen) return;
098:
099:         ITradingStorage.Trade memory t = o.trade;
100:         ITradingCallbacks.Values memory v = _openTradePrep
  (t, _a.current, o.maxSlippageP);
101:
102:         t.openPrice = uint64(v.priceImpact.priceAfterImpact);
103:
104:         if (v.cancelReason == ITradingCallbacks.CancelReason.NONE) {
105:             // Return excess counter trade collateral
106:             if (v.collateralToReturn > 0) {
107:                 TradingCommonUtils.transferCollateralTo
  (t.collateralIndex, t.user, v.collateralToReturn);
108:                 t.collateralAmount = v.newCollateralAmount;
109:                 emit ITradingCallbacksUtils.CounterTradeCollateralReturned
  (_a.orderId, t.collateralIndex, t.user, v.collateralToReturn);
110:             }
111:
112:             t = _registerTrade(t, o, _a.current);
```

The problem is that inside the `_openTradePrep` function the `priceAfterImpact` (== open price) is firstly calculated on the initial (full) collateral amount of the trade (L698) and only after that `newCollateralAmount` (reduced) amount is calculated in case if the trade is `counter` (L730):

```
File: TradingCallbacksUtils.sol
688:     function _openTradePrep(
689:         ITradingStorage.Trade memory _trade,
690:         uint256 _executionPriceRaw,
691:         uint256 _maxSlippageP
692:     ) internal  returns (ITradingCallbacks.Values memory v) {

694:             _trade.collateralAmount,
695:             _trade.leverage
696:         );
697:
698:         v.priceImpact = TradingCommonUtils.getTradeOpeningPriceImpact(
699:             ITradingCommonUtils.TradePriceImpactInput
  (_trade, _executionPriceRaw, positionSizeCollateral, 0),
700:             _getMultiCollatDiamond().getCurrentContractsVersion()
701:         );
702:
703:         uint256 maxSlippage = (uint256
  (_trade.openPrice) * _maxSlippageP) / 100 / 1e3;
704:
...
728:
729:         if
  (_trade.isCounterTrade && v.cancelReason == ITradingCallbacks.CancelReason.NONE) {
730:             (
  v.cancelReason,
  v.collateralToReturn,
  v.newCollateralAmount
) = _validateCounterTrade(
731:                 _trade,
732:                 positionSizeCollateral
733:             );
734:         }
735:     }
```

This means that `counter` trades with excess collateral would face unfavorable open prices since the last ones would always be calculated based on the full collateral amount instead of actual reduced amounts.

The same problem exists in the `executeTriggerOpenOrderCallback` since it also utilizes `_openTradePrep`.

Recommendations

Consider updating the `TradingCallbacksUtils#_openTradePrep` function so it firstly calculates `newCollateralAmount` (without excess collateral that would be returned to the user) and only after calculates the price impact based on this value instead of the initial (full) collateral amount.

[M-06] Incorrect LINK fee is charged for the `PNL_WITHDRAWAL` order type

Severity

Impact: Medium

Likelihood: Medium

Description

The `TradingInteractionsUtils.withdrawPositivePnl` function uses a half of `TradingCommonUtils.getMinPositionSizeCollateral(t.collateralIndex, t.pairIndex)` (minimum position size in collateral tokens for a pair) value as a `GetPriceInput.positionSizeCollateral` parameter. This parameter is used for the LINK fee calculation in the `PriceAggregatorUtils.getLinkFee` function. But the `getLinkFee` function always uses the maximum between the `_positionSizeCollateral` and the minimum position size in collateral tokens for a pair. So an actual LINK fee will always be higher than expected.

```
function withdrawPositivePnl(
      uint32_index,
      uint120_amountCollateral
    ) external tradingActivatedOrCloseOnly {
<...>
        _getMultiCollatDiamond().getPrice(
            IPriceAggregator.GetPriceInput(
                t.collateralIndex,
                t.pairIndex,
                ITradingStorage.Id({user: t.user, index: t.index}),
                orderId,
                pendingOrder.orderType,
>>              TradingCommonUtils.getMinPositionSizeCollateral
  (t.collateralIndex, t.pairIndex) / 2,
                ChainUtils.getBlockNumber(),
                t.isCounterTrade
            )
        );
```

```
function getLinkFee(
<...>
        uint256 linkFeeCollateral = _getMultiCollatDiamond
          ().pairOraclePositionSizeFeeP(_pairIndex) *
            feeRateMultiplier *
            TradingCommonUtils.getPositionSizeCollateralBasis(
                _collateralIndex,
                _pairIndex,
>>              _positionSizeCollateral,
                feeRateMultiplier
            ); // avoid stack too deep
```

TradingCommonUtils.sol

```
function getPositionSizeCollateralBasis(
        uint8 _collateralIndex,
        uint256 _pairIndex,
        uint256 _positionSizeCollateral,
        uint256 _feeRateMultiplier
    ) public view returns (uint256) {
>>      uint256 minPositionSizeCollateral = getMinPositionSizeCollateral
  (_collateralIndex, _pairIndex);
        uint256 adjustedMinPositionSizeCollateral =
          (minPositionSizeCollateral * 1e3) / _feeRateMultiplier;
        return
            _positionSizeCollateral > adjustedMinPositionSizeCollateral
                ? _positionSizeCollateral
                : adjustedMinPositionSizeCollateral;
    }
```

Recommendations

Consider dividing `linkFeePerNode` by 2 when `_input.orderType == ITradingStorage.PendingOrderType.PNL_WITHDRAWAL`:

```
function getPrice(
        IPriceAggregator.GetPriceInput memory _input
    ) external validCollateralIndex(_input.collateralIndex) {
<...>
        // 2. Calculate link fee for each oracle
        TradingCommonUtils.updateFeeTierPoints
          (_input.collateralIndex, _input.tradeId.user, _input.pairIndex, 0);
        uint256 linkFeePerNode = getLinkFee(
            _input.collateralIndex,
            _input.tradeId.user,
            _input.pairIndex,
            _input.positionSizeCollateral,
            _input.isCounterTrade
        ) / s.oracles.length;
+
+       if
+ (_input.orderType == ITradingStorage.PendingOrderType.PNL_WITHDRAWAL) {
+           linkFeePerNode = linkFeePerNode / 2;
+       }
```

```
function withdrawPositivePnl(
      uint32_index,
      uint120_amountCollateral
    ) external tradingActivatedOrCloseOnly {
<...>
        _getMultiCollatDiamond().getPrice(
            IPriceAggregator.GetPriceInput(
                t.collateralIndex,
                t.pairIndex,
                ITradingStorage.Id({user: t.user, index: t.index}),
                orderId,
                pendingOrder.orderType,
-               TradingCommonUtils.getMinPositionSizeCollateral
- (t.collateralIndex, t.pairIndex) / 2,
+               TradingCommonUtils.getMinPositionSizeCollateral
+ (t.collateralIndex, t.pairIndex),
                ChainUtils.getBlockNumber(),
                t.isCounterTrade
            )
        );
```

8.4. Low Findings

[L-01] SetRoles prevents self-assignment for `GOV_TIMELOCK` only

The setRoles function is designed to allow only accounts with either the `GOV_TIMELOCK` or `GOV_EMERGENCY_TIMELOCK` role to assign roles to other accounts. To prevent privilege escalation, the function should not allow a caller to assign themselves the `GOV_TIMELOCK` or `GOV_EMERGENCY_TIMELOCK` role. However, the current implementation only checks for self-assignment of the `GOV_TIMELOCK` role:

```
if (_roles[i] == Role.GOV_TIMELOCK && _accounts[i] == msg.sender) {
    revert NotAllowed();
}
```

There is no equivalent check for `Role.GOV_EMERGENCY_TIMELOCK`, even though the function is callable by that role as well.

Recommendation:

```
if (
    (_roles[i] == Role.GOV_TIMELOCK && _accounts[i] == msg.sender) ||
    (_roles[i] == Role.GOV_EMERGENCY_TIMELOCK && _accounts[i] == msg.sender)
) {
    revert NotAllowed();
}
```

[L-02] State update in `executeManualHoldingFeesRealizationCallback` may cause fee errors

The `executeManualHoldingFeesRealizationCallback` function calls `resetTradeBorrowingFees` to reset a trade's borrowing fee:

```
function executeManualHoldingFeesRealizationCallback(
        ITradingStorage.PendingOrder memory _order,
        ITradingCallbacks.AggregatorAnswer memory _a
    ) external {
        ...
        _getMultiCollatDiamond().resetTradeBorrowingFees(
            trade.collateralIndex,
            trade.user,
            trade.pairIndex,
            trade.index,
            trade.long,
            _a.current
        );
        ...
    }
```

The `resetTradeBorrowingFees` function calculates current accumulated fees using `getBorrowingPairPendingAccFees` and stores them as the trade's new initialAccFees:

```
function resetTradeBorrowingFees(
        uint8 _collateralIndex,
        address _trader,
        uint16 _pairIndex,
        uint32 _index,
        bool _long,
        uint256 _currentPairPrice
    ) public validCollateralIndex(_collateralIndex) {
        uint256 currentBlock = ChainUtils.getBlockNumber();

        (
          uint64pairAccFeeLong,
          uint64pairAccFeeShort,
          ,

        ) = getBorrowingPairPendingAccFees(
            _collateralIndex,
            _pairIndex,
            currentBlock,
            _currentPairPrice
        );
        ...

        _getStorage
          ().initialAccFees[_collateralIndex][_trader][_index] = initialFees;

        ...
    }
```

The issue is that this function only updates the trade-specific initialAccFees but does not update the newly calculated accumulated fee back to the global state (`borrowingFeesStorage.pairs`). And the `pairAccFeeLong/pairAccFeeShort` is now affected by the current pair price.

Consider the scenario:

- Manual realization at high price: `executeManualHoldingFeesRealizationCallback` is called when the pair price is high (e.g., 100e10).
- Trade baseline updated: The trade's `initialAccFees` is set based on accumulated fees calculated at the high price.
- Global state remains stale: The pair's global accumulated fees (BorrowingFeesStorage.pairs.accFeeLong/accFeeShort) are NOT updated.
- Price drops significantly: The pair price drops to a much lower value (e.g., 50e10).
- Subsequent fee calculation: When borrowing fees are calculated again, the current global accumulated fees (calculated at the lower price) may be lower than the stored `initialAccFees`.
- Underflow revert: The calculation `current global accumulated fees - initialAccFees` underflows, causing transaction reverts.

It's recommended to update global accumulated fees when `executeManualHoldingFeesRealizationCallback` is called.

[L-03] Admin can change price impact parameters for active trades

The `PriceImpactUtils.setUserPriceImpact` internal function allows a privileged admin to set custom `cumulVolPriceImpactMultiplier` and `fixedSpreadP` for specific trader-pair combinations. The issue with the current implementation is the ability to alter these parameters for a user's `already open positions`, changing the rules of the game post-entry. It can lead to worsens the effective closing price and liquidation price calculations.

Recommendation: It's recommended to modify the logic so that parameters set via `setUserPriceImpact` only apply to trades opened after the parameters are set for that user-pair.

[L-04] Rounding errors can favor users by a better liquidation price

In the `DecreasePositionSizeUtils` and `BorrowingFeesUtils` contracts, there are potential rounding errors that can result in users being favored by a better liquidation price. Specifically, in the `prepareCallbackValues` function of `DecreasePositionSizeUtils`, the calculation of `newLiqPrice` involves a division operation that can lead to rounding errors, then in the `getTradeLiquidationPrice` function of `BorrowingFeesUtils`, the calculation can also result in rounding errors. DecreasePositionSizeUtils.sol

```
function prepareCallbackValues(
        ITradingStorage.Trade memory _existingTrade,
        ITradingStorage.Trade memory _partialTrade,
        ITradingCallbacks.AggregatorAnswer memory _answer
    ) internal view returns
      (IUpdatePositionSizeUtils.DecreasePositionSizeValues memory values) {
<...>
        // 6. Calculate existing and new trade liquidation price
        values.existingLiqPrice = TradingCommonUtils.getTradeLiquidationPrice
          (_existingTrade, _answer.current);
        values.newLiqPrice = TradingCommonUtils.getTradeLiquidationPrice(
            _existingTrade,
            _existingTrade.openPrice,
            values.newCollateralAmount,
            values.newLeverage,
            values.isLeverageUpdate ? int256(
              values.closingFeeCollateral
            ) - values.pnlToRealizeCollateral : int256(0
            _getMultiCollatDiamond().getTradeLiquidationParams
              (_existingTrade.user, _existingTrade.index),
            _answer.current,
            values.isLeverageUpdate
                ? 1e18
>>              : (
  (values.existingPositionSizeCollateral - values.positionSizeCollateralDelta) * 1e18) /
                    values.existingPositionSizeCollateral,
            false
        );


        /// @dev Other calculations are in separate helper called after
        // realizing pending holding fees
        // Because they can be impacted by pending holding fees realization
        //(eg. available in diamond calc)
    }
```

BorrowingFeesUtils.sol

```
function getTradeLiquidationPrice
      (IBorrowingFees.LiqPriceInput calldata _input) internal view returns (uint256) {

            _input.collateralIndex,
            address(0), // never apply fee tiers
            _input.pairIndex,
            (_input.collateral * _input.leverage) / 1e3,
            _input.isCounterTrade
        );


        IFundingFees.TradeHoldingFees memory holdingFees;
        int256 totalRealizedPnlCollateral;


        if (!_input.beforeOpened) {
            holdingFees = _getMultiCollatDiamond
              ().getTradePendingHoldingFeesCollateral(
                _input.trader,
                _input.index,
                _input.currentPairPrice
            );


            (, , totalRealizedPnlCollateral) = _getMultiCollatDiamond
              ().getTradeRealizedPnlCollateral(
                _input.trader,
                _input.index
            );
        }


        uint256 spreadP = _getMultiCollatDiamond().pairSpreadP
          (_input.trader, _input.pairIndex);
        ITradingStorage.ContractsVersion contractsVersion = _input.beforeOpened
            ? _getMultiCollatDiamond().getCurrentContractsVersion()
            : _getMultiCollatDiamond().getTradeContractsVersion
              (_input.trader, _input.index);


        return
            _getTradeLiquidationPrice(
                _input.openPrice,
                _input.long,
                _input.collateral,
                _input.leverage,
                int256(closingFeesCollateral) +
                    (
                      (holdingFees.totalFeeCollateral - totalRealizedPnlCollateral) *
>>                      int256(_input.partialCloseMultiplier)) /
                    1e18 +
                    _input.additionalFeeCollateral,
                _getMultiCollatDiamond().getCollateral
                  (_input.collateralIndex).precisionDelta,
                _input.liquidationParams,
                contractsVersion,
                spreadP
            );
    }
```

Consider rounding the `_feesCollateral` value towards positive infinity.

[L-05] `updateFeeTierPoints` is not properly called during several operations

`getTradeClosingPriceImpact` is a function that calculates the price impact for a specific trade based on multiple factors, one of which is the total fee in the collateral token

(`getTotalTradeFeesCollateral`).

```
function getTradeClosingPriceImpact(
        ITradingCommonUtils.TradePriceImpactInput memory _input
    ) public view returns (
      ITradingCommonUtils.TradePriceImpactmemoryoutput,
      uint256tradeValueCollateralNoFactor
    ) public view returns
      (ITradingCommonUtils.TradePriceImpact memory output, uint256 tradeValueCollateralNoFactor

        // ...

        tradeValueCollateralNoFactor = getTradeValueCollateral(
            trade,
            getPnlPercent(
                trade.openPrice,
                getPriceAfterImpact(
                    _input.oraclePrice,

                                        output.fixedSpreadP + cumulVolPriceImpactP + output.skewI
                ),
                trade.long,
                trade.leverage
            ),
>>>         getTotalTradeFeesCollateral(
                trade.collateralIndex,
                trade.user,
                trade.pairIndex,
                getPositionSizeCollateral
                  (trade.collateralAmount, trade.leverage),
                trade.isCounterTrade
            ),
            _input.currentPairPrice
        );
// ...
}
```

`getTotalTradeFeesCollateral` calculates the fee using the trader's fee multiplier cached in `calculateFeeAmount`. However, in several instances, `getTradeClosingPriceImpact` is called without updating the trade's fee points.

```
function getTotalTradeFeesCollateral(
        uint8 _collateralIndex,
        address _trader,
        uint16 _pairIndex,
        uint256 _positionSizeCollateral,
        bool _isCounterTrade
    ) public view returns (uint256) {
        uint256 feeRateP = _getMultiCollatDiamond().pairTotalPositionSizeFeeP
          (_pairIndex);

        uint256 feeRateMultiplier = _isCounterTrade
            ? _getMultiCollatDiamond().getPairCounterTradeFeeRateMultiplier
              (_pairIndex)
            : 1e3;

        uint256 rawFeeCollateral = (getPositionSizeCollateralBasis(
            _collateralIndex,
            _pairIndex,
            _positionSizeCollateral,
            feeRateMultiplier
        ) * feeRateP) /
            ConstantsUtils.P_10 /
            100 /
            1e3;

        // Fee tier is applied on min fee, but counter trade fee rate multiplier
        // doesn't impact min fee
>>>     return _getMultiCollatDiamond().calculateFeeAmount
  (_trader, rawFeeCollateral);
    }
```

Here is the list of functions:

`executeManualNegativePnlRealizationCallback` :

```
function executeManualNegativePnlRealizationCallback(
        ITradingStorage.PendingOrder memory _order,
        ITradingCallbacks.AggregatorAnswer memory _a
    ) external {
        ITradingStorage.Trade memory trade = _getTrade
          (_order.trade.user, _order.trade.index);

        if (!trade.isOpen) return;

        (
          ITradingCommonUtils.TradePriceImpactmemorypriceImpact,

        ) = TradingCommonUtils.getTradeClosingPriceImpact(
            ITradingCommonUtils.TradePriceImpactInput(
                trade,
                _a.current,
                TradingCommonUtils.getPositionSizeCollateral
                  (trade.collateralAmount, trade.leverage),
                _a.current,
                false // don't use cumulative volume price impact
            )
        );
// ..
}
```

`closeTradeMarketCallback` :

```
function closeTradeMarketCallback(
        ITradingCallbacks.AggregatorAnswer memory _a
    ) external tradingActivatedOrCloseOnly {
        ITradingStorage.PendingOrder memory o = _getPendingOrder(_a.orderId);

        _validatePendingOrderOpen(o);

        ITradingStorage.Trade memory t = _getTrade(o.trade.user, o.trade.index);
        ITradingStorage.TradeInfo memory i = _getTradeInfo
          (o.trade.user, o.trade.index);

        (
          ITradingCommonUtils.TradePriceImpactmemorypriceImpact,

        ) = TradingCommonUtils.getTradeClosingPriceImpact(
            ITradingCommonUtils.TradePriceImpactInput(
                t,
                _a.current,
                TradingCommonUtils.getPositionSizeCollateral
                  (t.collateralAmount, t.leverage),
                _a.current,
                true
            )
        );
// ...
}
```

`executeTriggerCloseOrderCallback =>` validateTriggerCloseOrderCallback` :

```
function validateTriggerCloseOrderCallback(
        ITradingStorage.Id memory _tradeId,
        ITradingStorage.PendingOrderType _orderType,
        uint64 _open,
        uint64 _high,
        uint64 _low,
        uint64 _currentPairPrice
    ) public view returns
      (ITradingStorage.Trade memory t, ITradingCallbacks.Values memory v) {
       // ...
        // Return early if trade is not open
        if (v.cancelReason != ITradingCallbacks.CancelReason.NONE) return
          (t, v);
        v.liqPrice = TradingCommonUtils.getTradeLiquidationPrice
          (t, _currentPairPrice);

                uint256 triggerPrice = _orderType == ITradingStorage.PendingOrderType.TP_CLOSE
            ? t.tp
            :
              (_orderType == ITradingStorage.PendingOrderType.SL_CLOSE ? t.sl : v.liqPrice);


                v.exactExecution = triggerPrice > 0 && _low <= triggerPrice && _high >= triggerPr
        v.executionPriceRaw = v.exactExecution ? triggerPrice : _open;


        if (_orderType != ITradingStorage.PendingOrderType.LIQ_CLOSE) {
>>>         (v.priceImpact, ) = TradingCommonUtils.getTradeClosingPriceImpact(
                ITradingCommonUtils.TradePriceImpactInput(
                    t,
                    v.executionPriceRaw,
                    TradingCommonUtils.getPositionSizeCollateral
                      (t.collateralAmount, t.leverage),
                    _currentPairPrice,
                    true
                )
            );
            v.executionPrice = v.priceImpact.priceAfterImpact;
        } else {
            v.executionPrice = v.executionPriceRaw;
        }

        // ...
    }
```

Recommendations

Consider adding or moving the `updateFeeTierPoints` calls to the start of the operation.

[L-06] MillisecondsPerBlock value for Ape chain is outdated

In the `ChainUtils.convertBlocksToSeconds` function, a hardcoded value of 1000 milliseconds (1 second) per block is used for Ape chain. However, according to current network metrics, the average block time on Ape chain is approximately 1.5 seconds per block, not 1 second.

```
function convertBlocksToSeconds(uint256 _blocks) internal view returns
      (uint256) {
        uint256 millisecondsPerBlock;
        ...
        } else if (block.chainid == APECHAIN_MAINNET) {
            millisecondsPerBlock = 1000; // @audit 1500 miliseconds
        ...

        return Math.mulDiv
          (_blocks, millisecondsPerBlock, 1000, Math.Rounding.Up);
    }
```

It's recommended to update the `millisecondsPerBlock` value for Ape chain to 1500 milliseconds to match the current network's average block time

Reference: https://apescan.io/chart/blocktime

[L-07] Borrowing fees not accrued before updating fee per block cap parameters

When admins update global or pair-specific borrowing fee cap parameters via `setBorrowingFeePerBlockCap` or `setBorrowingPairFeePerBlockCapArray`, the protocol fails to accrue spending borrowing fees based on the previous parameters before applying the changes.

```
function setBorrowingFeePerBlockCap
      (IBorrowingFees.BorrowingFeePerBlockCap memory _feePerBlockCap) external {
        _validateBorrowingFeePerBlockCap(_feePerBlockCap);

        _getStorage
        //().feePerBlockCap = _feePerBlockCap; // @audit missing fee accrual before update
        ...
    }

    function setBorrowingPairFeePerBlockCapArray(
        uint8 _collateralIndex,
        uint16[] calldata _indices,
        IBorrowingFees.BorrowingFeePerBlockCap[] calldata _values
    ) external validCollateralIndex(_collateralIndex) {
            ...
        for (uint256 i; i < len; ++i) {
            IBorrowingFees.BorrowingFeePerBlockCap memory value = _values[i];
            _validateBorrowingFeePerBlockCap(value);

            uint16 pairIndex = _indices[i];
            _getStorage
            //().pairFeePerBlockCaps[_collateralIndex][pairIndex] = value; // @audit missing fee a

            ...
        }
    }
```

This issue causes incorrect fee calculation for the period spanning from the last accrual to the parameter update, as the new parameters are retroactively applied to this period. Traders may be either overcharged or undercharged borrowing fees depending on whether the new cap parameters are higher or lower than the previous ones.

Before updating fee cap parameters, accrue pending borrowing fees to ensure that historical fees are calculated correctly.

[L-08] Less burn amount if negative PnL is realized early

When negative PnL is realized manually through the `executeManualNegativePnlRealizationCallback` function, the losses are sent to the GToken vault but are not burned since they're considered not final:

```
function executeManualNegativePnlRealizationCallback(
        ITradingStorage.PendingOrder memory _order,
        ITradingCallbacks.AggregatorAnswer memory _a
    ) external {
            ...
                TradingCommonUtils.getGToken
                  (trade.collateralIndex).receiveAssets(
                    negativePnlToRealizeCollateral,
                    trade.user,
                    false // don't burn unrealized pnl since it's not final
                );
            ...
    }
```

However, when the trade is eventually closed, the protocol doesn't account for the previously realized negative PnL (which finalized at this point). Because the manually realized negative PnL has already reduced the `availableCollateralInDiamond`, only the remaining negative PnL (if any) is sent to the GToken vault with the burn flag set to true:

```
function handleTradeValueTransfer(
        ITradingStorage.Trade memory _trade,
        int256 _collateralSentToTrader,
        int256 _availableCollateralInDiamond
    ) external {
            ...
        } else if (_collateralSentToTrader < _availableCollateralInDiamond) {
            // Send loss to gToken
            getGToken(_trade.collateralIndex).receiveAssets(
                uint256
                  (_availableCollateralInDiamond - _collateralSentToTrader),
                _trade.user,
                true // any amount sent to vault at this point is negative PnL,
                // funding fees already sent with _burn = false
            );
        }
    }
```

This leads to a discrepancy in burn amounts between two scenarios:

- With manual negative PnL realization:
  - X1 amount is sent to GToken but NOT burned via manual realization.
  - Later, X2 amount is sent to GToken and burned when the trade is closed.
  - Total burned: X2.
- Without manual negative PnL realization:
  - The trade is closed with the full amount (X1 + X2) sent to GToken and burned.
  - Total burned: X1 + X2.

This issue undermines the protocol's tokenomics by reducing the amount of collateral eligible for burning.

When a trade is closed, the protocol should account for previously realized negative PnL in the burn calculation.

[L-09] Position size validation uses pre-fee amounts

The protocol incorrectly validates position sizes using pre-fee amounts rather than the actual position size after fees are deducted (`totalPositionSizeFeeP`). This affects two key areas:

1. Counter trade collateral return:

The calculated `exceedingPositionSizeCollateral` is too high because `_positionSizeCollateral` represents the pre-fee amount. This results in returning more collateral than intended to counter trade users.

```
function validateCounterTrade(
        ITradingStorage.Trade memory _trade,
        uint256 _positionSizeCollateral
    ) external view returns
      (bool isValidated, uint256 exceedingPositionSizeCollateral) {
        ...
        if (_positionSizeCollateral > maxPositionSizeCollateral)
            exceedingPositionSizeCollateral = _positionSizeCollateral -
            // maxPositionSizeCollateral; // @audit Pre-fee amount

        return (true, exceedingPositionSizeCollateral);
    }
```

It means counter trade users receive more collateral back than they should based on actual position sizes

2. Exposure limit validation:

The exposure limit check uses the pre-fee position size, which could cause valid trades to be incorrectly rejected when the post-fee position would actually be within limits.

For example, the `newSkewCollateral` is less than `maxSkewCollateral` if the opening fees are deducted from the collateral. However, because opening fees are not counted, the opening trade transaction reverts.

```
function _openTradePrep(
        ITradingStorage.Trade memory _trade,
        uint256 _executionPriceRaw,
        uint256 _maxSlippageP
    ) internal view returns (ITradingCallbacks.Values memory v) {
        ...
        v.cancelReason = (
                ...
                    : !TradingCommonUtils.isWithinExposureLimits(
                        _trade.collateralIndex,
                        _trade.pairIndex,
                        _trade.long,
                        positionSizeCollateral // @audit Pre-fee amount
                    )
                ...
    }
```

It leads to valid trades may be rejected due to exposure limit checks using inflated position sizes.

Calculate position sizes after fees before validating against limits.

[L-10] `ExecuteManualNegativePnlRealizationCallback` could revert on 0 transfer

Inside `executeManualNegativePnlRealizationCallback`, it calculates `realizedNegativePnlToCancelCollateral` and attempts to transfer the assets from the vault to the `GNSMultiCollatDiamond` contract.

```
function executeManualNegativePnlRealizationCallback(
        ITradingStorage.PendingOrder memory _order,
        ITradingCallbacks.AggregatorAnswer memory _a
    ) external {
        ITradingStorage.Trade memory trade = _getTrade
          (_order.trade.user, _order.trade.index);

        if (trade.isOpen) {

                trade,
                _a.current
            );

                        uint256 unrealizedNegativePnlCollateral = unrealizedPnlCollateral < 0
                ? uint256(-unrealizedPnlCollateral)
                : uint256(0);


                .getTradeManuallyRealizedNegativePnlCollateral
                  (trade.user, trade.index);

                        uint256 newManuallyRealizedNegativePnlCollateral = existingManuallyRealiz

            if
              (unrealizedNegativePnlCollateral > existingManuallyRealizedNegativePnlCollateral) {
                // ...


                                newManuallyRealizedNegativePnlCollateral += negativePnlToRealizeC
            } else {
>>>
               uint256 realizedNegativePnlToCancelCollateral = existingManuallyRealizedNegativePnl
                    unrealizedNegativePnlCollateral;

>>>             TradingCommonUtils.receiveCollateralFromVault(
                    trade.collateralIndex,
                    realizedNegativePnlToCancelCollateral
                );


                                newManuallyRealizedNegativePnlCollateral -= realizedNegativePnlTo
            }

            _getMultiCollatDiamond().storeManuallyRealizedNegativePnlCollateral(
                trade.user,
                trade.index,
                newManuallyRealizedNegativePnlCollateral
            );

            emit ITradingCallbacksUtils.TradeNegativePnlManuallyRealized(
                _a.orderId,
                trade.collateralIndex,
                trade.user,
                trade.index,
                unrealizedNegativePnlCollateral,
                existingManuallyRealizedNegativePnlCollateral,
                newManuallyRealizedNegativePnlCollateral,
                _a.current
            );
        }

        _getMultiCollatDiamond().closePendingOrder(_a.orderId);
    }
```

However, it doesn't account for scenarios where `realizedNegativePnlToCancelCollateral` is 0. If the collateral transfer reverts on a 0 transfer, the operation will fail. Only trigger `receiveCollateralFromVault` when `realizedNegativePnlToCancelCollateral` is greater than 0.

[L-11] Reverted request IDs stay pending for the protocol

The `ChainlinkClientUtils.validateChainlinkCallback` function deletes pending requests IDs from the protocol storage, i.e. makes them invalid. But in case of revert due to invalid candles the protocol storage returns in the previous state. So the request ID stays pending. The Chainlink protocol has similar validations but in turn it ignores errors. Consider avoiding reverts in case of invalid candles and using early return instead.

```
function fulfill(bytes32 _requestId, uint256 _priceData) external {
>>      ChainlinkClientUtils.validateChainlinkCallback(_requestId);

        IPriceAggregator.PriceAggregatorStorage storage s = _getStorage();

        IPriceAggregator.Order memory order = s.orders[_requestId];
        ITradingStorage.Id memory orderId = ITradingStorage.Id
          ({user: order.user, index: order.index});


                IPriceAggregator.OrderAnswer[] storage orderAnswers = s.orderAnswers[orderId.user
        uint8 minAnswers = s.minAnswers;
        bool usedInMedian = orderAnswers.length < minAnswers;

        if (usedInMedian) {
            IPriceAggregator.OrderAnswer memory newAnswer;
            (
              newAnswer.current,
              newAnswer.open,
              newAnswer.high,
              newAnswer.low,
              newAnswer.ts
            ) = _priceData
                .unpackAggregatorAnswer();

            // Valid inputs:
            // 1. non-lookback:
            // - current > 0 => open/high/low ignored
            // 2. lookback:
            // - open > 0, high > 0, low > 0
            //(high >= open, low <= open), current > 0
            if (
                newAnswer.current == 0 ||
                (order.isLookback &&
                    (newAnswer.high < newAnswer.open ||
                        newAnswer.low > newAnswer.open ||
                        newAnswer.open == 0 ||
                        newAnswer.low == 0))
>>          ) revert IPriceAggregatorUtils.InvalidCandle();
```

[L-12] Rounding errors causes minimal position size less than expected

The `TradingCommonUtils.getPositionSizeCollateralBasis` function determines a minimal position size to use when charging fees by comparing `_positionSizeCollateral` with `adjustedMinPositionSizeCollateral`, i.e. `minPositionSizeCollateral` value being adjusted by dividing on `_feeRateMultiplier`. The problem is the return value will be multiplied by `_feeRateMultiplier` again, so there can be unnecessary rounding loss

when `minPositionSizeCollateral` is divided and then multiplied by `_feeRateMultiplier`.

```
function getPositionSizeCollateralBasis(
        uint8 _collateralIndex,
        uint256 _pairIndex,
        uint256 _positionSizeCollateral,
        uint256 _feeRateMultiplier
    ) public view returns (uint256) {
        uint256 minPositionSizeCollateral = getMinPositionSizeCollateral
          (_collateralIndex, _pairIndex);
>>      uint256 adjustedMinPositionSizeCollateral =
  (minPositionSizeCollateral * 1e3) / _feeRateMultiplier;
        return
            _positionSizeCollateral > adjustedMinPositionSizeCollateral
                ? _positionSizeCollateral
                : adjustedMinPositionSizeCollateral;
    }
```

Consider avoiding division by `_feeRateMultiplier`:

```
function getPositionSizeCollateralBasis(
        uint8 _collateralIndex,
        uint256 _pairIndex,
        uint256 _positionSizeCollateral,
        uint256 _feeRateMultiplier
    ) public view returns (uint256) {
        uint256 minPositionSizeCollateral = getMinPositionSizeCollateral
          (_collateralIndex, _pairIndex);
-       uint256 adjustedMinPositionSizeCollateral =
- (minPositionSizeCollateral * 1e3) / _feeRateMultiplier;
+       minPositionSizeCollateral = minPositionSizeCollateral * 1e3;
+
+        uint256 adjustedPositionSizeCollateral = _positionSizeCollateral * _feeRateMultiplier;
        return
-           _positionSizeCollateral > adjustedMinPositionSizeCollateral
-               ? _positionSizeCollateral
-               : adjustedMinPositionSizeCollateral;
+           adjustedPositionSizeCollateral > minPositionSizeCollateral
+               ? adjustedPositionSizeCollateral
+               : minPositionSizeCollateral;
    }
```

PriceAggregatorUtils.sol

```
function getLinkFee(
<...>
        uint256 linkFeeCollateral = _getMultiCollatDiamond
          ().pairOraclePositionSizeFeeP(_pairIndex) *
-           feeRateMultiplier *
            TradingCommonUtils.getPositionSizeCollateralBasis(
                _collateralIndex,
                _pairIndex,
                _positionSizeCollateral,
                feeRateMultiplier
            );
```

TradingCommonUtils.sol

```
function getTotalTradeFeesCollateral(
        uint8 _collateralIndex,
        address _trader,
        uint16 _pairIndex,
        uint256 _positionSizeCollateral,
        bool _isCounterTrade
    ) public view returns (uint256) {
        uint256 feeRateP = _getMultiCollatDiamond().pairTotalPositionSizeFeeP
          (_pairIndex);

        uint256 feeRateMultiplier = _isCounterTrade
            ? _getMultiCollatDiamond().getPairCounterTradeFeeRateMultiplier
              (_pairIndex)
            : 1e3;

        uint256 rawFeeCollateral = (getPositionSizeCollateralBasis(
            _collateralIndex,
            _pairIndex,
            _positionSizeCollateral,
            feeRateMultiplier
        ) *
-           feeRateP *
-           feeRateMultiplier) /
+           feeRateP /
            ConstantsUtils.P_10 /
            100 /
            1e3;
```