Bitperp Audit v2

H1 `checkTp` Reverts When Order Direction is Sell

The `maxGainP` parameter is set to **900**, which implies a maximum leverage of **9x** for buy orders.

During order creation (`openOrder`) or when updating take-profit (`updateTp`), the `checkTp` function validates the take-profit (`tp`) value.

However, if the order direction is sell and `maxGainP` is set to the default value of 900, the following issue arises:

- For a leverage of 1, the `maxTpDist` (maximum take-profit distance) will be `9 * price` .
- The validation condition `tp < price && tp >= price - maxTpDist` will fail, causing the function to revert.

Impact:This results in a denial of service (DoS) for the `openOrder` function when the order direction is sell.

```
639
640     function checkTp(uint256 tp, uint256 price, uint256 leverage, bool buy) external view {
641 //      maxTp  9x
642 //      maxGainP 900
643         uint256 maxTpDist = price * maxGainP / 100 / leverage;
644         require(
645 //          :qa price - maxTpDist will revert , because  price< maxTpDist
646             tp == 0 || (buy ? (tp > price && tp <= price + maxTpDist) : (tp < price && tp >= price - maxTpDist)),
647             "WRONG_TP"
648         );
649     }
650
```

H2 Potential Issue of Uncleared Pending Orders

Some user operations involving the oracle can result in pending statuses.

Example: Legacy order trade opening:

Flow: `openTrade` → `storePendingMarketOrder`
Oracle → `openTradeMarketCallback` → `unregisterPendingMarketOrder`

One invariant is the pendingOrder count.
The issue arises when the oracle return callback execution order is not sequential.

Scenario:Assume a situation where a user's position is unhealthy, and simultaneously:

- An executor executes liquidation.
- The user executes closeTrade.

This creates a PendingMarketOrder in storage.

Problem: If `executeExecutorCloseOrderCallback` executes first (since it does not depend on the oracle's callback), it does not clear the pending status or unregister the order. Subsequently, when `closeTradeMarketCallback` is called, it will fail, leaving a permanently uncleared pending order in storage.

```
1  function closeTradeMarketCallback(AggregatorAnswer memory a) external
   onlyPriceAggregator notDone {
2  ...
3
4  storageT.unregisterPendingMarketOrder(a.orderId, false);
5
6
7  }
8
9
10
```

```
11 function executeExecutorCloseOrderCallback(
12 ...
13
14 }
```

When opening an order, the system enforces a pending order count check. This mechanism can inadvertently lead to a Denial of Service (DoS) for users, preventing them from initiating new trades or closing existing positions.

M1 Non-Legacy Orders Should Not Be Subject to Pending Order Limit

When opening a trade, the system enforces a pending order limit.
This check is applied to both non-legacy orders and pending orders.
However, only pending orders invoke `storePendingMarketOrder`, while non-pending orders are still unnecessarily subject to this limit.

```
1 function openTrade(
2 ...
3 require(storageT.pendingOrderIdsCount(sender) <
  storageT.maxPendingMarketOrders(), "MAX_PENDING_ORDERS");
4
5 if (orderType != StorageInterfaceV7.OpenLimitOrderType.LEGACY) {
6 ..
7
8 } else {
9 storageT.storePendingMarketOrder(
```

M2 Excessive `trade.leverage` or `pairOpenFeeP(pairIndex)` values can cause `openTradeMarketCallback` to fail

When the `registerTrade` method is called, the following logic calculates and deducts `v.reward2` (i.e., trading fees):

```
519         v.levPosDai = trade.positionSizeDai * trade.leverage;
520         v.tokenPriceDai = aggregator.tokenPriceDai();
521
522         // 2. Charge opening fee - referral fee (if applicable)
523         v.reward2 = storageT.handleDevGovFees(trade.pairIndex, v.levPosDai, true, true);
524
525         trade.positionSizeDai -= v.reward2;
```

The trading fee is specifically calculated through the `storageT.handleDevGovFees()` function.

```
515     // Manage dev & gov fees
516     function handleDevGovFees(uint256 _pairIndex, uint256 _leveragedPositionSize, bool _dai, bool _fullFe
517         external
518         onlyTrading
519         returns (uint256 fee)
520     {
521         fee = _leveragedPositionSize * priceAggregator.openFeeP(_pairIndex) / PRECISION / 100;
522         if (!_fullFee) fee /= 2;
523
524         if (_dai) {
525             govFeesDai += fee;
526             devFeesDai += fee;
527         } else {
528             govFeesToken += fee;
529             devFeesToken += fee;
530         }
531
532         fee *= 2;
533     }
```

Problem:If either `trade.leverage` or `pairOpenFeeP(pairIndex)` is excessively large, it will result in an abnormally high `v.reward2` (fee). This can cause the operation `trade.positionSizeDai -= v.reward2` to overflow, resulting in a failure of the `registerTrade()` method, which in turn causes the `openTradeMarketCallback` function to fail.

Current Constraints:

`trade.leverage` : Set by privileged roles, with a maximum limit of 1000.

`pairOpenFeeP(pairIndex)` : Also set by privileged roles, but currently has no defined upper limit.

```
125     modifier feeOk(Fee calldata _fee) {
126         require(
127             _fee.openFeeP > 0 && _fee.closeFeeP > 0 && _fee.referralFeeP > 0 && _fee.minLevPosDai > 0,
128             "WRONG_FEES"
129         );
130         _;
131     }
```

Example 1:

- `trade.leverage = 1000`
- `trade.positionSizeDai = 10 DAI`

- `PRECISION = 1e10`
- If `priceAggregator.openFeeP(_pairIndex)` exceeds 1,000,000, the operation `trade.positionSizeDai -= v.reward2` will overflow, leading to an error and causing `openTradeMarketCallback` to fail.

Example 2:

If the fee rate is 1%:

1. When `trade.positionSizeDai = 100` and `trade.leverage = 1`:
    - Fee calculation formula: Fee = positionSizeDai × leverage × fee rate × 2.
    - Fee: 100 × 1 × 0.01 × 2 = 2.0 DAI.
    - Remaining position size after deducting the fee: 100 - 2 = 98.0 DAI.
2. When `trade.positionSizeDai = 100` and `trade.leverage = 100`:
    - Fee: 100 × 100 × 0.01 × 2 = 200.0 DAI.
    - Remaining position size after deducting the fee: 100 - 200 = -100.0 DAI.

Summary:

- Case 1: The fee is reasonable, leaving a remaining position size of 98.0 DAI.
- Case 2: The fee exceeds the position size, resulting in a negative remaining position size of -100.0 DAI, triggering an overflow issue.

L *Referral Fee Never Charged*

The `ReferralFeeCharged` event is defined and commented in the code but is never actually triggered or used, indicating that referral fees are not being charged as intended.

```
1 event ReferralFeeCharged(address indexed trader, uint256 valueDai);
```

```
550
551         // Shared code between market & limit callbacks
552         function registerTrade(StorageInterfaceV7.Trade memory trade)
553             private
554             returns (StorageInterfaceV7.Trade memory, uint256)
555         {
556             AggregatorInterfaceV7 aggregator = storageT.priceAggregator();
557             PairsStorageInterfaceV7 pairsStored = aggregator.pairsStorage();
558
559             Values memory v;
560
561             v.levPosDai = trade.positionSizeDai * trade.leverage;
562             v.tokenPriceDai = aggregator.tokenPriceDai();
563
564 //              event ReferralFeeCharged(address indexed trader, uint256 valueDai);
565 //              :qa we have  ReferralFeeCharged event,and comment, but never charged  ,event not used
566             // 2. Charge opening fee - referral fee (if applicable)
567             v.reward2 = storageT.handleDevGovFees( uint256: trade.pairIndex,  uint256: v.levPosDai,  bool: true,  bool: true);
568
```