SHERLOCK

# Security Review For Avantis

Private Contest Prepared For: **Avantis**
Lead Security Expert: **bughuntoor**
Date Audited: **October 8 - November 1, 2024**

## Introduction

Avantis is the world's first fully onchain exchange focused on cross asset leverage (real world assets + crypto), where you'd see BTC-USD and USD-JPY trending on the same day! The contest focuses on their v1.5 launch, covering several new trading specific features such as CEX-like fee tiers and a new perp order type (variable PnL fee).

## Scope

Repository: Avantis-Labs/avantis-contracts

Branch: v1.5

Audited Commit: 881be832afed762dc5f8d3194fce36ed5feeb1c3

Final Commit: ee61312b28a2aea934cb8f9484afd7433e82b776

---

For the detailed scope, see the contest details.

## Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

## Issues Found

| High | Medium |
|---|---|
| 2 | 27 |

## Issues Not Fixed or Acknowledged

| High | Medium |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

eeyore
jokr
samuraii77
bughuntoor
KupiaSec
TurnipBoy
Ironsidesec
Varun_05
Afriaudit
aslanbek
qpzm
thekmj
pseudoArtist
jah
1nc0gn170

# Issue H-1: Any referrer can overwrite other traders' referral codes into their own, stealing rebates and decreasing their discount

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/3

## Found by

1nc0gn170, Afriaudit, Ironsidesec, KupiaSec, bughuntoor, eeyore, jah, jokr, qpzm, samuraii77, thekmj

## Summary

Due to wrong access control/wrong design of referral code usage, any referrer can overwrite any other traders' referral codes into their own. This redirects all trading fee rebates to the adversary's account instead of the rightful referrer, and will even cause traders to lose funds due to forced decreased discount.

## Root Cause

In `Referral`, the function `setTraderReferralCodeByOwner()` allows anyone (any referrer) to overwrite any trader's code into their own code:

```
function setTraderReferralCodeByOwner(bytes32 _code, address _trader) external
↪   whenNotPaused{
    require(codeOwners[_code] == msg.sender, "OWNER_ONLY");
    _setTraderReferralCode(_trader, _code); // @audit-note anyone can do this
}
```

https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/Referral.sol#L170

Anyone can also become a referrer just by registering their own code of choice, as long as they haven't own any:

```
function registerCode(bytes32 _code) external whenNotPaused {
    require(_code != bytes32(0), "Referral: invalid _code");
    require(codeOwners[_code] == address(0), "Referral: code already exists");
    require(codes[msg.sender] == bytes32(0),"Referral: referrer already
↪   registered");

    codeOwners[_code] = msg.sender;
    codes[msg.sender] =  _code;
```

```
    referrerTiers[msg.sender] = _DEFAULT_TIER_ID;
    isPrivate[_code]  = false;

    emit RegisterCode(msg.sender, _code);
}
```

https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/Referral.sol#L180

When a trade is opened, the trading fee discount is applied to the trader, and the trading fee rebate is sent to the referrer:

```
function applyReferralAndPnlFee(
    address _trader,
    uint _fees,
    uint _leveragedPosition,
    bool _isPnl,
    uint _pairIndex,
    int _percentProfit,
    uint _collateral
) public override onlyTrading returns (uint, uint) {
    // ...
    (uint traderFeePostDiscount, address referrer, uint referrerRebate) =
↪   referral.traderReferralDiscount(_trader, _fees);

    rebates[referrer] += referrerRebate;
}
```

https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/TradingStorage.sol#L580-L582

The referrer can easily claim the rebate through `claimRebate()`, without the admin having any power to stop them.

https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/TradingStorage.sol#L649-L656

## Internal pre-conditions

None

## External pre-conditions

None

## Attack Path

1. Alice (whale) deposits a large amount of margin. Alice applies a tier-3 code to get the maximum fee discount of 15%.
2. Bob monitors the mainnet, and sees this activity.
3. Bob creates a new referral code, and overwrites Alice's code into their own with `setTraderReferralCodeByOwner()`. New referral codes starts at tier 1, given only a 5% discount.
4. When Alice opens a trade, the fee discount/rebate is applied based on Bob's code instead.

If Alice sets back the old code, Bob can simply backrun her to set his new code again. The end result is that:

- Alice only gets a 5% fee discount, instead of the rightful 15% from Alice's referrer.
- Alice's referrer loses the 15% fee rebate completely. Bob gets a 5% fee rebate from Alice's trade.

## Impact

- Attacker gets all the trading rebates for themselves. Legit referrers lose that trading rebate.
- Trader is forced a lower fee discount with the attacker's tier-1 referral code, instead of being able to take a higher discount with a tier-3 referral code that the trader wills to.

## PoC

*No response*

## Mitigation

Traders should be allowed to decide which referral code they wish to use. There are some possible designs for this:

- Delete `setTraderReferralCodeByOwner()` entirely. For private codes, a referrer should only be able to invite a trader to use their code. The trader should explicitly accept the invitation to use the code.
- `setTraderReferralCodeByOwner()` should only be possible if the trader hasn't been applied any referral codes. The trader should be allowed to change code at will, if the current referral code isn't from whom they know.

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/54/

# Issue H-2: Incorrect collateral value for re-balance if partial trade converts to full trade

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/130

## Found by

Ironsidesec, KupiaSec, TurnipBoy, bughuntoor, eeyore, jokr

## Summary

In the `_unregisterTrade` function, after a trade is partially or fully closed, the open interest reserved for the trade is released:

```
storageT.vaultManager().releaseBalance(_collateral.mul(_trade.leverage));
```

For a partial trade, `_collateral` represents the amount of collateral the trader wants to close, while for a full trade close, it equals the initialPosToken of the trade.

However, in some cases, when a trader partially closes a trade, any remaining dust collateral is also considered closed. This approach is intended to manage insignificant positions.

```
if (_trade.initialPosToken == _collateral || (_collateral + totalFees >=
↪  _trade.initialPosToken)){
    storageT.unregisterTrade(_trade.trader, _trade.pairIndex, _trade.index);
    pairInfos.resetTradeInitialAccess(_trade.trader, _trade.pairIndex,
↪  _trade.index);
    _collateral = _trade.initialPosToken;
}
else {
    storageT.registerPartialTrade(_trade.trader, _trade.pairIndex, _trade.index,
↪  _collateral);
}
```

The problem arises because in such cases, the rebalance should consider the complete `initialPosToken`, not just the provided `_collateral` for the partial trade. As a result, the open interest corresponding to the remaining part is not released, even though the trade is effectively closed.

This unreleased open interest accumulates and becomes unrecoverable, compounding the issue due to two factors:

1. High leverage trades amplify the open interest value that remains stuck.

2. For large trades where total fees on partial closes are high, the seized collateral is also high. This leads to a significantly higher open interest value being stuck. Due to this LPs would not be able to withdraw funds in some cases as utilisation ratio would go down gradually

## Root Cause

- In TradingCallbacks.sol:554 the `releaseBalance` function uses incorrect collateral amount when a partial trade is made but full trade is closed

## Internal pre-conditions

*No response*

## External pre-conditions

*No response*

## Attack Path

1. A trader opens a long position in a BTC pair with the following parameters:
   - Collateral: 5000 USDC
   - Leverage: 100x
   - Opening price: 50,000 USDC
2. Reserved amount = 5000 * 100 = 500,000 USDC
3. The BTC price then rises to 55,000, allowing the trader to achieve a maximum profit of 500%.
4. The trader decides to partially close the trade, withdrawing 4600 USDC.
   - Total fees for the partial trade amount to approximately 420 USDC.
   - Rebalance amount = 4600 * 100 = 460,000 USDC
5. After the partial trade of 4600 USDC, the remaining 400 USDC collateral is less than the total fees of 420 USDC. Therefore, the remaining collateral will also be closed, and the trade will be registered as a full close.
6. In this scenario, the rebalance should release the full 500,000 USDC, but it only rebalances 460,000 USDC.
7. As a result, the stuck reserved open interest is:

```
stuck reserved open interest = 500,000 - 460,000 = 40,000 USDC
```

## Impact

This causes significant amount of open interest to become unreservable, making it unavailable for other trades. LPs would not be able to withdraw funds as utilisation ratio would go down

## PoC

*No response*

## Mitigation

If the trade is converted from partial to full, then release the full amount

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/42/files

# Issue M-1: Loss protection inconsistency might result in users taking higher losses than supposed to.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/29

The protocol has acknowledged this issue.

## Found by

bughuntoor, jokr, qpzm

## Summary

In skewed market, users can have loss protection of 10%-20%. Based on the provided docs, the loss protection should work as follows:

> Example 2 (Directional Trader): If the net PnL of the above trader is -20%, then technically the trader should have lost $2,000 (20% of $10K). However, with loss protection, the trader only loses $2000 * (1-20%) = $1,600. This is the magic of loss protection !

However, the current implementation works slightly different.

```
function getTradeValuePure(
    uint collateral,
    int percentProfit,
    uint rolloverFee,
    uint closingFee,
    uint lossProtection
) public view returns (uint, int, uint) {
    int pnl = (int(collateral) * percentProfit) / int(_PRECISION) / 100;
    int lossProtectedPnl = pnl;
    if (pnl < 0) {
        lossProtectedPnl = (pnl * int(lossProtection)) / 100;
    }

    int fees = int(rolloverFee) + int(closingFee);
    int value = int(collateral) + lossProtectedPnl - fees;
    if (value <= (int(collateral) * int(100 - liqThreshold)) / 100) {
        value = lossProtectedPnl - pnl;
        lossProtectedPnl = fees - int(collateral) + value;
    }
    return (value > 0 ? uint(value) : 0, lossProtectedPnl, uint(fees));
}
```

We can see that if the value is less than 15% of the collateral, the value paid out to the user is the delta of the `lossProtectedPnl-pnl`.

Consider the following situation:

1. User has opened a $10k position with loss protection of 10%.
2. Position's current pnl is -95%. This would make the `lossProtectedPnl` equal to 85.5%. For simplicity we ignore the margin/ rollover fees.
3. This would cause us to enter the if-statement. Then the value assigned to the position would be `-$8550-(-9500)=$950`.
4. According to the example above, the position's value should've been `$10000-0.9*(9500)=$1450`. Instead it is just $950, resulting in $450 extra loss to the user. This is an extra loss of $450, or the user receiving ~35% less than supposed to.

## Root Cause

Wrong calculation of position's value

## Affected Code

https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/PairInfos.sol#L671

## Impact

Loss of funds

## Mitigation

Fix is non-trivial

# Issue M-2: Wrong handling of wallet open interest will cause issues

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/61

The protocol has acknowledged this issue.

## Found by

samuraii77

## Summary

Wrong handling of wallet open interest will cause issues

## Root Cause

Not properly handling wallet open interest per each pair.

Each wallet can have a certain amount of max interest (the code below is a check in `TradingStorage::withinExposureLimits()` that must hold true when opening a trade):

```
walletOI(_trader) + _leveragedPos <= pairsStored.maxWalletOI(_pairIndex);
```

The issue is that we are checking the total wallet open interest of a trader against the max wallet open interest for a particular pair. Let's take a look at `PairStorage::maxWalletOI()`:

```
function maxWalletOI(uint _pairIndex) external view override returns (uint) {
    return (storageT.maxOpenInterest() * pairs[_pairIndex].values.maxWalletOI) /
↪   100;
}
```

As seen, we calculate the max open interest for a wallet based on the pair index provided. For the `walletOI` function used in the check above, we can clearly see that there is no pair index input, thus we can see that the return value of the function is not based on the pair of the trade. Let's also take a look at `TradingStorage::_updateOpenInterestUSDC()` where the wallet open interest for a trader gets updated:

```
_walletOI[_trader] = _open ? _walletOI[_trader] + _leveragedPosUSDC :
↪   _walletOI[_trader] - _leveragedPosUSDC;
```

As seen, it gets updated the same way regarding of the pair index.

From the docs, I could not deduct whether the intentions were for each pair to have its own max wallet OI or for there to be a max wallet OI across all pairs. Either way, the current implementation is wrong as it is a mixture of the 2.

## Internal pre-conditions

*No response*

## External pre-conditions

*No response*

## Attack Path

1. There are 2 pairs, each with a maximum open interest of 100
2. Bob creates a trade on one of the pairs with a position size of 100
3. His wallet open interest gets updated to 100
4. He tries to create another trade on the other pair with a position size of 100
5. This reverts as he already has 100 wallet open interest which is also the maximum wallet open interest for the other pair (despite him having 0 open trades on the other pair)

## Impact

Unexpected reverts

## PoC

*No response*

## Mitigation

Refactor the code to match your intention (either max wallet open interest for each pair or max wallet open interest across all pairs)

# Issue M-3: Precision loss in `VaultManager#getCollateralFee` will make users overpay `balancingFee`

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/64

The protocol has acknowledged this issue.

## Found by

aslanbek

## Summary

On deposits and withdrawals, a balancing fee may or may not be charged, depending on the current reserve ratio in tranches:

1. Tranche#deposit -> ERC4626#previewDeposit -> Tranche#_deposit -> getDepositFeesTotal -> balancingFee; Tranche#withdraw -> ERC4626#previewWithdraw -> Tranche#_withdraw -> getWithdrawalFeesRaw -> balancingFee
2. Tranche#balancingFee -> VaultManager#getBalancingFee

https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/VaultManager.sol#L415-L429

As we can see, instead of bps, `getDynamicReserveRatio` returns percents.

Therefore, if the actual reserve ratio is 6299 bps, it would return 62.

in `getBalancingFee`:

```
if ((getDynamicReserveRatio(tranche, isDeposit, assets) * 100) >
↪  balancingDeltaThreshold) {
```

62*100 < 6250 => `getBalancingFee` returns 0 instead of 500, despite the actual reserve ratio being outside of the target range

so instead of paying the balancingFee because actual ratio is above 6250 (value from initializer), users will not.

Similarly, when the ratio is between 3751 and 3799 bps, `getDynamicReserveRatio` would return 37, and `balancingFee` return a fee of 500, even though the current reserve ratio is within the target range.

## Root Cause

Percentage precision instead of bps in `getDynamicReserveRatio`.

## Internal pre-conditions

getDynamicReserveRatio is in 3750-3799 or 6750-6799 bps range

## External pre-conditions

Deposit or withdrawal into a Tranche.

## Impact

Deposits and withdrawals pay [do not pay] the `balancingFee` they shouldn't [should].

## PoC

**Case 1:**

juniorTranche has 62_999e6 USDC seniorTranche has 37_101e6 USDC

Alice wants to withdraw 100e6 from seniorTranche ratio = 62999e6 * 100 / 100_000e6 = 62

62*100 = 6200

6200 < 6250

`balancingFee` is not charged (but it should be)

**Case 2:**

seniorTranche has 62_001e6 USDC juniorTranche has 38_099e6 USDC

Alice wants to withdraw 100e6 from juniorTranche ratio = 37_999e6 * 100 / 100_000e6 = 37

37*100 = 3700

3700 < 3750

`balancingFee` is charged (but it should not be)

## Mitigation

`getDynamicReserveRatio` should return bps instead of percentage; its return value should not be multiplied by `100` in `getBalancingFee` anymore.

# Issue M-4: Fee distribution will be DOS'd when utilizationRatio >= withdrawThreshold for either tranche

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/72

## Found by

TurnipBoy, bughuntoor

## Summary

Whenever the vaultManager distributes fees it attempts to call _distributeCollectFeeShares.

VaultManager.sol#L642-L652

```
function _distributeCollectedFeeShares(address _tranche) internal {
    uint256 assets = ITranche(_tranche).redeem( <- @audit attempts to redeem
        ITranche(_tranche).maxRedeem(address(this)),
        address(this),
        address(this)
    );

    if (assets > 0) {
        _distributeVeRewards(IVeTranche(ITranche(_tranche).veTranche()), assets);
    }
}
```

We see above that it will always attempt to redeem from the tranche.

Tranche.sol#L288-L305

```
function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal virtual override {
    uint256 fee = getWithdrawalFeesRaw(assets);
    super._withdraw(caller, receiver, owner, assets, shares);

    if (fee > 0) {
        SafeERC20.safeTransfer(ERC20(asset()), address(vaultManager), fee);
```

```
        vaultManager.allocateRewards(fee, false);
        emit FeeTransferredToVM(fee, false);
    }
    require(utilizationRatio() < withdrawThreshold, "UTILIZATION_RATIO_MAX"); <-
    ↪  @audit reverts even when withdrawing 0
}
```

The problem here is that L304 reverts when utilizationRatio() < withdrawThreshold, hence all distributions will fail.

## Root Cause

VaultManager:L643 always redeems shares even if maxRedeem = 0

## Internal pre-conditions

utilizationRatio >= withdrawThreshold

## External pre-conditions

None

## Attack Path

1. Fees are ready to be distributed
2. utilizationRatio >= withdrawThreshold for one of the tranches
3. Fee distributions will always revert

## Impact

Fee distribution will be DOS'd

## PoC

*No response*

## Mitigation

Redeem should be skipped if maxRedeem returns as 0

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/37

# Issue M-5: Guaranteed SL, forced trade durations and pnl fee structure leads to pairs of delta neutral trades that always make money

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/77

The protocol has acknowledged this issue.

## Found by

TurnipBoy

## Summary

The following exploit hinges on 3 factors:

1. Stop losses are guaranteed to execute even if the current price is below the SL price
2. Trades over a certain amount CANNOT be closed before a certain amount of time
3. PNL fee structure makes all fees scale with profit rather than being fixed

Trading.sol#L512-L522

```
if (_orderType == ITradingStorage.LimitOrder.LIQ) {
    uint liqPrice =  pairInfos.getTradeLiquidationPrice(
                        t.trader,
                        t.pairIndex,
                        t.index,
                        t.openPrice,
                        t.buy,
                        t.initialPosToken,
                        t.leverage
                    );
    require(t.sl == 0 || (t.buy ? liqPrice > t.sl : liqPrice < t.sl), "HAS_SL"); <-
     ↪  will revert because sl price is better than liquidation price
```

We see above that liquidation cannot be called unless the liquidation price is worse than the sl price. This means that as long as we have a stop loss open we can never be liquidated under that price. This means that we must execute as a stop loss limit to liquidate a position with a valid stop loss.

Trading.sol#L523-L525

```
}else{
    require(block.timestamp - t.timestamp >=
    ↪  pairsStored.openCloseThreshold(t.pairIndex,
    ↪  t.initialPosToken.mul(t.leverage)), "EARLY_CLOSE");
}
```

This leads us to the next point. Stop losses cannot be triggered until at least a certain amount of time has passed for the largest position sizes, this is 15 minutes. This give us 15 free minutes in which we cannot be liquidated.

TradingCallbacks.sol#L454-L463

```
_trade.positionSizeUSDC -= isPnl
                           ? 0 <-@audit no open fee
                           : storageT.handleDevGovFees(
                             _trade.trader,
                             _trade.pairIndex,
                             _trade.positionSizeUSDC.mul(_trade.leverage),
                             true,
                             true,
                             _trade.buy
                           );
```

PairStorage.sol#L608-L618

```
function getPnlBasedFee(uint pairIndex, uint collateral, int percentProfit)
 ↪  external view override returns(uint) {
    if(percentProfit < 0) return 0; // No Fee charged for losses

    uint i = 0;
    for(i; i < fees[pairIndex].pnlFees.numTiers; i++){
        if(uint(percentProfit) < fees[pairIndex].pnlFees.tierP[i]) break;
    }

    uint pnl = collateral*uint(percentProfit)/ _PRECISION/ 100;
    return fees[pairIndex].pnlFees.feesP[i] * pnl/ _PRECISION / 100;
}
```

We see for pnl trades that the open fee is 0 and that exit fees are applied only on the profit made on the trade.

Together these three small issues lead to a large problem. A trade can place two opposing trades, one long and one short. Each trade has a stop loss with tolerance just small enough for the stop loss to be valid. For 15 minutes neither of these trades can be liquidated or closed. During this time however, the underlying asset will move either up or down even by a small amount. After 15 minutes, one trade will be liquidated for virtually no loss and the other can be closed for a profit.

Reporting as HIGH since currently pairIndex 34 (ORDI) and 35 (STX) are hardcoded to behave in this manner and can be exploited from the moment this upgrade goes into effect.

## Root Cause

1. Trading.sol:L522 guarantees that stop losses must be used instead of liquidation for all crypto pairs (including ORDI and STX)
2. Trading.sol:L524 guarantees that stop losses cannot be executed for 15 minutes after the order is place
3. PairStorage.sol:L574 enforces open close timer for pairIndex 34 (ORDI) and 35 (STX)
4. TradingCallbacks.sol:L455 charges no open fee
5. PairStorage.sol:617 charges pnl fees proportional to gains and are not fixed

## Internal pre-conditions

None

## External pre-conditions

Target asset price is different by any amount in 15 minutes

## Attack Path

1. Open a long and short PNL market order with extremely tight stop loss with max leverage
2. Wait 15 minutes
3. Allows one order to trigger stop loss with minute loss
4. Close the other order for guaranteed profit

## Impact

All funds can be drained from LPs

## PoC

Add the following to the test/units/ folder to show how these delta neutral trades can make guaranteed money:

```
contract GuaranteedSLExploit is TradeBase {

    function testSLExploit() public {
    // open long
        uint rand = uint(keccak256(abi.encodePacked(block.timestamp))) % numTraders;

        vm.startPrank(traders[rand]);

        uint amount = usdc.balanceOf(traders[rand]) / 2;
        usdc.approve(address(tradingStorage), amount );

        uint id = _placeMarketLong(traders[rand], amount, btcPairIndex,  50000);
        vm.stopPrank();
        _executeMarketLong(traders[rand], amount, btcPairIndex,  50000, id);

    // open short
        vm.startPrank(traders[rand]);

        usdc.approve(address(tradingStorage), amount );

        uint id2 = _placeMarketShort(traders[rand], amount, btcPairIndex,  50000);
        vm.stopPrank();
        _executeMarketShort(traders[rand], amount, btcPairIndex,  50000, id2);

    // roll block forwards to set SL since I cant find method in test suite to set
    ↪  sl during order creation
        vm.roll(5);

        vm.startPrank(traders[rand]);
        trading.updateTpAndSl{value:
        ↪  mockPyth.getUpdateFee(_generateSampleUpdateDataCrypto(1, btcPairIndex,
        ↪  50000))}(
            btcPairIndex,
            traderOrderIndex[traders[rand]] -1,
            withPricePrecision(50050),
            withPricePrecision(49500),
            _generateSampleUpdateDataCrypto(1, btcPairIndex, 50000)
            );

        vm.stopPrank();
        vm.startPrank(operator);

        uint256 price = 50500;

        _setChainlinkBTC(price);

    // losing trade cannot be liquidated
        _executeLimitOrderThatWillFail(
            ITradingStorage.LimitOrder.LIQ,
```

```
            traders[rand],
            btcPairIndex,
            traderOrderIndex[traders[rand]] -1,
            price);

        vm.stopPrank();

    // fast forward 15 minutes
        vm.warp(900);

    // stop loss short
        vm.startPrank(operator);

        _executeLimitOrder(
            ITradingStorage.LimitOrder.SL,
            traders[rand],
            btcPairIndex,
            traderOrderIndex[traders[rand]] -1,
            price);
        vm.stopPrank();

    // close long for profit
        vm.startPrank(traders[rand]);
        _trade =
            tradingStorage.openTrades(traders[rand], btcPairIndex, 0);

        uint closed = _placeMarketClose(btcPairIndex, _trade.initialPosToken, 0,
        ↪  price);
        vm.stopPrank();
        _executeMarketClose(btcPairIndex, _trade.initialPosToken,
        ↪  traderOrderIndex[traders[rand]] - 1,  price, closed);

    // show that trade has made money
        assert(usdc.balanceOf(traders[rand]) > amount * 2);
    }

    function _executeLimitOrderThatWillFail(
        ITradingStorage.LimitOrder _orderType,
        address _trader,
        uint _pairIndex,
        uint _index,
        uint _rawPrice) internal{

        _setChainlinkBTC(_rawPrice);

        bytes[] memory pythData = _generateSampleUpdateDataCrypto(1, btcPairIndex,
        ↪  _rawPrice);

        uint fee = mockPyth.getUpdateFee(pythData);
```

```
        vm.expectRevert();
        trading.executeLimitOrder{value: fee}(
            _orderType,
            _trader,
            _pairIndex,
            _index,
            pythData);
    }

    function _executeLimitOrder(
        ITradingStorage.LimitOrder _orderType,
        address _trader,
        uint _pairIndex,
        uint _index,
        uint _rawPrice) internal{

        _setChainlinkBTC(_rawPrice);

        trading.executeLimitOrder{value:
        ↪  mockPyth.getUpdateFee(_generateSampleUpdateDataCrypto(1, btcPairIndex,
        ↪  _rawPrice))}(
            _orderType,
            _trader,
            _pairIndex,
            _index,
            _generateSampleUpdateDataCrypto(1, btcPairIndex, _rawPrice ));
    }
}
```

## Mitigation

Guaranteed SL and forced trade duration should never be enabled on the same pair. ORDI and STX markets should be amended here to remove their hard coded forced trade duration.

# Issue M-6: `VaultManager` accounting is off and will result in significant errors

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/84

The protocol has acknowledged this issue.

## Found by

Ironsidesec, bughuntoor, eeyore

## Summary

Whenever a user has to receive funds for their payout, the `VaultManager` only makes sure to hold enough funds to cover for the currently accrued borrowing fees.

```
function _sendUSDCToTrader(address _trader, uint _amount) internal {

    // For the extereme case of totalRewards exceeding vault Manager balance
    int256 balanceAvailable = int(storageT.usdc().balanceOf(address(this))) -
↪   int(totalRewards);
    if (int(_amount) > balanceAvailable) {
        // take difference (losses) from vaults
        uint256 difference = uint(int(_amount) - int(balanceAvailable));

        uint256 seniorUSDC = (getSeniorLossMultiplier() * difference * (100 -
↪   getReserveRatio(0)))/100 / 100;
        seniorUSDC = (seniorUSDC > difference) ? difference : seniorUSDC;

        uint256 juniorUSDC =  difference - seniorUSDC;
        junior.withdrawAsVaultManager(juniorUSDC);
        senior.withdrawAsVaultManager(seniorUSDC);
    }

    require(storageT.usdc().transfer(_trader, _amount));
    emit USDCSentToTrader(_trader, _amount);
}
```

This means that in case the contract holds other funds, it will first use them, instead of the tranche funds. Such funds that might be within the contract are:

- Traders' collateral
- Liquidators' execution fees.

Using either of these funds will result in unfair results for the liquidity providers.

Consider the following scenarios:

**Scenario 1**

1. There are only 2 traders, both with $1000 position size, one of them is +100% pnl and the other one is -100% pnl. Borrowing fees and pnl fees accrued currently within the contract are currently 0. (for simplicity, we'll ignore borrowing fees)
2. The positive pnl trader closes their position first. Since there are enough funds within the VaultManager (due to the other trader's collateral), the funds are simply transferred to the trader. None of the tranche's share values are impacted, although they should be (as pnlrewards are 0).
3. When the negative pnl position is closed, the user has nothing to receive, but the `VaultManager` will add 1000 USDC to its pnlRewards. Note that contract will now not hold any funds and there's nothing to actually back up these pnl rewards.

**Scenario 2**

1. There are only 2 traders, both with $1000 position size, both are at +100% pnl. Borrowing fees and pnl fees accrued currently within the contract are currently 0. (for simplicity, we'll ignore borrowing fees)
2. One of them closes their position. Since there are enough funds within the contract, the position will be instantly paid out and no funds will be taken from any of the tranches and therefore their share value will not be decreased (although they should).
3. After some time the the other trader closes their position too (assuming at same pnl). Now, since the contract will not hold any funds, it will take all of the payout from the tranche's ($2000). This would mean that the funds will be taken based on the tranche ratios at the closure of the 2nd position, when instead they $1000 should've been taken based on the ratio at the closure of first position and only the other $1000 at the closure of the 2nd position.

Note that any liquidity providers adding liquidity inbetween steps 2 and 3 will add at an unfairly overvalued tranche shares, which would be -EV action. Any LPs aware of the following issue, can withdraw inbetween steps 2 and 3 to avoid a loss.

## Root Cause

Wrong accounting

## Affected Code

https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/VaultManager.sol#L607

## Impact

Tranche share values will often be significantly off and not correspond to their true value. Tranche payouts will happen based at wrong ratios, which are not the ones at the time a position was closed. Possibly, there would be no funds to cover for accrued pnlfees.

## Mitigation

Implement global accounting for position collaterals and execution fees within the VaultManager

# Issue M-7: Liquidation might occur instantly after withdrawing max possible margin

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/105

## Found by

jokr

## Summary

An incorrect position health check after withdrawing margin could lead to the position becoming liquidatable immediately after the withdrawal.

The trader can withdraw margin up to an amount such that the effective position, after accounting for PnL, should be greater than 20% of the collateral. This ensures the position still has a 5% buffer to avoid reaching the liquidation state after withdrawing the maximum possible margin.

```
require((int(_trade.initialPosToken) + pnl) > (int(_trade.initialPosToken) *
↪  int(100 - _WITHDRAW_THRESHOLD_P)) / 100, "W_T_B");
```

However, this check fails in cases where the trader has loss protection. Since the PnL used in the above check is the loss-protected PnL, the trader is able to withdraw more margin than intended based on the loss protection.

But because the PnL used in the liquidation check does not consider loss protection, if the trader withdraws the maximum possible margin, their position will enter a liquidatable state immediately after the withdrawal.

## Root Cause

- In TradingCallbacks:85 margin withdrawal health check uses lossProtectedPnl, inflating position value and allowing excessive collateral than intended withdrawal while appearing healthy but actually not based on the liquidation price

## Internal pre-conditions

1. Trader should have pnl protection

## External pre-conditions

1. Trader should withdraw margin from open trade

## Attack Path

The trader opens a long position with the following parameters:

```
Collateral = 250 USDC
Leverage = 4x
LevPositionToken = 1000
Loss protection rebate = 20%

The price then drops by 9%, reducing the price to 910 USDC.

profitP = -9% * 4 = -36%
pnl = -36% of 250 = -90 USDC

net position value = 250 - 90 = 160
```

So the position remains in a healthy state.

Next, the trader tries to withdraw 150 USDC from the margin.

```
newCollateral = 100 USDC
newLeverage = 10x

profitP = -9% * 10 = -90%
pnl = -90% of 100 = -90 USDC
lossProtectedPnl = -72 USDC (20% loss protection rebate applied)

net position value = initialPosToken - lossProtectedPnl = 28
```

In this case, the health check passes since 28 is greater than 20% of newCollateral. The withdrawal is successful.

However, on checking if the account is liquidatable:

```
profitP = -90%
pnl = -90% of 100 = -90 USDC
net position value = 100 - 90 = 10
```

The position is now liquidated immediately because the net position value is less than 15% of the collateral.

## Impact

The health check does not work as intended in cases with loss protection, leading to traders potentially being liquidated after withdrawing margin.

## PoC

*No response*

## Mitigation

Consider using pnl instead on loss protected pnl in the health check

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/47

# Issue M-8: The trader might get liquidated when the position is closed due to inconsistent liquidation check

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/106

The protocol has acknowledged this issue.

## Found by

bughuntoor, jokr

## Summary

In certain cases, the trader could get liquidated when attempting to close their position, resulting in a portion of their collateral being seized.

A trader's position will be liquidated if their PnL loss exceeds 85% of the collateral. When the trader closes the position, several fees are incurred, such as closing fees, margin fees, and limit close fees (only if the user has set take-profit or stop-loss limits).

When a trader's position is liquidated or closed, the following check in `getTradeValuePure` determines the action based on the position's value.

```
if (value <= (int(collateral) * int(100 - liqThreshold)) / 100) {
    value = lossProtectedPnl - pnl;
    lossProtectedPnl = fees - int(collateral) + value;
}
```

This check ensures that if the position is liquidated, the user receives nothing (if there is no loss rebate). If the position is closed, they receive the remaining value after accounting for PnL and fees.

However, the above check calculates value as int value = int(collateral) + lossProtectedPnl - fees, where fees represent the total fees:

```
total fees = closing fee + margin fee + limit close fee (if the trader opts for TP
↪  or SL)
```

The trader should be liquidated when their position loss exceeds 85% of the collateral.

Suppose the trader closes their position with a loss of x% of the collateral. The total fees incurred for closing the position is y.

This issue occurs when:

```
y > 85% of collateral - x% of collateral
```

## Root Cause

- In PairInfos.sol:671 the check for liquidation compares the position's remaining value (collateral minus PnL and total fees) to the liquidation threshold (85% of collateral). However, this threshold does not include the effect of these fees, effectively lowering the trader's actual position value relative to the threshold.

## Internal pre-conditions

*No response*

## External pre-conditions

*No response*

## Attack Path

1. If a user closes their position with a loss of 84% of the collateral and the total fees amount to 2% of the collateral, then the position's value is calculated as follows:

```
value = collateral - pnl - total fees
value = collateral - 84% of collateral - 2% of collateral = 14% of collateral
```

2. Since the position value is less than 15% of the collateral, the condition is met and code block mentioned above would be executed, causing the trade to be liquidated. Their collateral is seized, and they receive no remaining value.

## Impact

Trader gets liquidated if he tries to close his position loosing a part of his collateral value

## PoC

*No response*

## Mitigation

Update liquidation check to account for total fees while comparision

# Issue M-9: PriceAggregator uses the same heartbeat for all the chainlink feeds

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/115

The protocol has acknowledged this issue.

## Found by

Ironsidesec, bughuntoor, jokr, samuraii77

## Summary

The `fulfill` function in `PriceAggregator.sol` function uses the same heartbeat (chainlinkValidityPeriod) to check the freshness of the price for all the price feeds. The problem with this is different price feeds have different heartbeats. Since they use the same heartbeat the heartbeat needs to be slowest of all of them or else the staleness check will fail most of the time. The issue is that if we use slowest heartbeat of all the feeds as `chainlinkValidityPeriod` it would allow the consumption of potentially very stale data.

## Root Cause

In PriceAggregator.sol:126 and PriceAggregator.sol:158 same heartbeat value `chainlinkValidityPeriod` is used to check the freshness of all price feeds.

## Internal pre-conditions

No pre-conditions

## External pre-conditions

No external pre-conditions

## Attack Path

1. Lets take BTC-USD feed and EUR-USD feed as examples.
2. On Base BTC-USD chainlink feed's heartbeat is 20 minutes and EUR-USD feed's heartbeat is 24 hours.
3. So If price is not changed beyond deviation threshold they will only be updated after heartbeat period is passed.

4. If we use 20 minutes as heartbeat period to check freshness of prices staleness check will fail for EUR-USD will fail most of the times because its heartbeat is 24 hours it doesn't get updated every 20 mins.
5. If we use 24 hours as heartbeat we might consume stale BTC-USD price because the actual heartbeat of that price feed is 20 mins and chainlink sets it heartbeat to 20 mins because BTC is volatile.
6. So two different heartbeat periods should be used for validating these two price feeds.

## Impact

1. Stale prices will be confirmed or
2. Price staleness check will fail most of the times.

## PoC

*No response*

## Mitigation

Use separate heartbeat periods for every price feed

# Issue M-10: status of referral code is always set to private irrespective of the provided bool

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/120

## Found by

Afriaudit, KupiaSec, jokr

## Summary

In `updateReferral` function in `Referral.sol` is setting the status of the referral code to true irrespective of the passed value. But when false is passed it should change the status of the referral code to false.

## Root Cause

In `updateReferral` function in Referral.sol:200 is setting the status of the referral code always to true irrespective of the passed boolean value.

## Internal pre-conditions

None

## External pre-conditions

None

## Attack Path

1. Handler first set the status of Referral code `ALICE` to private by calling `updateReferral(ALICE,true)`.
2. Now after some time handler needs to set the status of the code from private to normal for some reason.
3. Now handler calls `updateReferral(ALICE,false)` but status of the code will not be set to public because `updateReferral` is always setting the status of code to true irrespective of the value passed.

## Impact

Referral code statuses cannot be set to public from private.

## PoC

*No response*

## Mitigation

Instead of setting to true set the provided bool

```
    // @audit-issue setting to true irrespective of bool
    function updateReferral(bytes32 _code, bool _isPrivate) external
 ↪  onlyGovOrHandler {
        require(_code != bytes32(0), "Referral: invalid _code");
        require(codeOwners[_code] != address(0), "Referral: Code Owner Does Not
 ↪  Exist");

-       isPrivate[_code] = true;
+       isPrivate[_code] = _isPrivate;

        emit CodeUpdated(_code, _isPrivate);
    }
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/38

# Issue M-11: Stop loss orders cannot be liquidated even if the position is liquidatable

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/133

The protocol has acknowledged this issue.

## Found by

jokr, samuraii77

## Summary

Due to a check that requires the liquidation price to be greater than the Stop Loss price (in the case of a buy) in the `executeLimitOrder` function of the `Trading.sol` contract, the operator cannot liquidate an order if the stop loss is higher than the liquidation price, even when the trade is liquidatable due to sudden price moves or gaps. This check should only apply to guaranteed Stop Loss (SL) orders. Currently, however, it is also applied to non-guaranteed orders, making them unliquidatable in these cases.

## Root Cause

Incorrect check in Trading.sol:522

```
require(t.sl==0||(t.buy?liqPrice>t.sl:liqPrice<t.sl),"HAS_SL");
```

This check should only be done for pairs with guranteed stop loss.

## Internal pre-conditions

Gap down or gap up or sudden price move

## External pre-conditions

*No response*

## Attack Path

1. Trader open XYZ long at 500.
2. Set his stop loss to 450 and his liquidation price is 400.
3. XYZ is a commodity which got opened at a gap down price the later day at 395.

4. The traders position is now liquidatable because current price of XYZ is less than the liquidation price of 400. But it cannot be liquidated by the operator because of the above mentioned check and it can only be closed using a stop-loss limit close.

## Impact

Positions can’t be liquidated even if they are in the liquidation stage.

## PoC

*No response*

## Mitigation

Do that check only for stop loss guaranteed stop loss pairs

# Issue M-12: No slippage protection in the `closeTradeMarket()` function.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/141

The protocol has acknowledged this issue.

## Found by

Ironsidesec, KupiaSec, eeyore, jokr

## Summary

When users call the `closeTradeMarket()` function, a new `Close` pending market order is created. This pending market order can only be finalized by the `Operator` role. Since this action may be delayed, users can be negatively impacted by market conditions.

This would be acceptable if the user action was fully within their control, such as when they control transaction gas usage.

However, as users do not control when this pending action will be completed and are unable to cancel it, they are exposed to potential fund loss.

Combining this with the fact that the protocol supports a `guaranteed` stop loss price, which is not respected in the case of market orders, users may experience greater losses than expected if the execution is delayed.

## Root Cause

No slippage protection in the `closeTradeMarket()` function flow; the `slippageP` and `wantedPrice` parameters of the PendingMarketOrder struct are set to 0. (here)

Even if set, these parameters are not used within the `TradingCallbacks.closeTradeMarketCallback()` function. (here)

## Internal pre-conditions

1. User has an open position in one of the `guaranteed` stop loss markets.
2. User sets the stop loss at 10% below the position open price.

## External pre-conditions

1. There are external negative market conditions, and the user expects these to worsen over time.

2. The current loss is only 5% and is still above the `guaranteed` stop loss.
3. User calls `closeTradeMarket()`, expecting the trade to close soon.

## Attack Path

1. User calls `closeTradeMarket()` while facing a 5% loss, expecting the trade to close shortly.
2. The Operator call is delayed, and by the time it is executed, the market has declined further, resulting in a 12% loss.
3. The `guaranteed` stop loss at 10% is not respected, causing the user to lose an additional 2% beyond the expected stop loss.

## Impact

- User experiences a loss of funds beyond his control, which cannot be seen as user mistake.

## PoC

Add to `MarketTrade.t.sol`, and call `forgetest-vv--match-testtest_PocMarketCloseNoSlippage`:

```
function test_PocMarketCloseNoSlippage() public {
    vm.startPrank(traders[0]);
    usdc.transfer(traders[2], usdc.balanceOf(traders[0]));
    uint amount = 500e6;
    usdc.mint(traders[0], amount);
    usdc.approve(address(tradingStorage), amount);

    uint id = _placeMarketLong(traders[0], amount, btcPairIndex,  50000);
    vm.stopPrank();
    _executeMarketLong(traders[0], amount, btcPairIndex,  50000, id);

    vm.startPrank(traders[1]);
    usdc.transfer(traders[2], usdc.balanceOf(traders[1]));
    usdc.mint(traders[1], amount);
    usdc.approve(address(tradingStorage), amount);

    id = _placeMarketLong(traders[1], amount, btcPairIndex,  50000);
    vm.stopPrank();
    _executeMarketLong(traders[1], amount, btcPairIndex,  50000, id);

    vm.roll(3);

    vm.startPrank(traders[0]);
```

```
    trading.updateTpAndSl{value:
↪   mockPyth.getUpdateFee(_generateSampleUpdateDataCrypto(1, btcPairIndex, 50000))}(
        btcPairIndex,
        0,
        withPricePrecision(49500),
        withPricePrecision(51000),
        _generateSampleUpdateDataCrypto(1, btcPairIndex, 50000)
        );

    vm.startPrank(traders[1]);
    trading.updateTpAndSl{value:
↪   mockPyth.getUpdateFee(_generateSampleUpdateDataCrypto(1, btcPairIndex, 50000))}(
        btcPairIndex,
        0,
        withPricePrecision(49500),
        withPricePrecision(51000),
        _generateSampleUpdateDataCrypto(1, btcPairIndex, 50000)
        );

    vm.warp(100000);
    _setChainlinkBTC(49750); // theoretical -5%

    ITradingStorage.Trade memory _trade =
        tradingStorage.openTrades(traders[0], btcPairIndex, 0);

    vm.startPrank(traders[0]);
    console.log("User decides to market close at ~ -5%");
    uint closeId = trading.closeTradeMarket(btcPairIndex, 0,
↪   _trade.initialPosToken); // trader 0 decides to market close at -5%
    vm.stopPrank();

    vm.warp(15);
    _setChainlinkBTC(49400); // theoretical ~12%
    vm.recordLogs();
    console.log("Operator triggers market close at ~ -12%, even though user SL is
↪   at ~ -10%");
    _executeMarketClose(btcPairIndex, 0, 0, 49400, closeId);

    Vm.Log[] memory entries = vm.getRecordedLogs();
    int percentProfit;
    (,,,,,percentProfit,,) = abi.decode(entries[17].data, (uint,
↪   ITradingStorage.Trade, bool, uint, uint, int, uint, bool));
    console.log("User 1, without slippage protection, experienced a funds loss of
↪   an extra 2% as his stop price was not triggered");
    console2.log("User 1 percentProfit:", percentProfit);

    vm.startPrank(operator);
    vm.recordLogs();
    trading.executeLimitOrder{value:
↪   mockPyth.getUpdateFee(_generateSampleUpdateDataCrypto(1, btcPairIndex, 49400))}(
```

```
        ITradingStorage.LimitOrder.SL,
        traders[1],
        btcPairIndex,
        0,
        _generateSampleUpdateDataCrypto(1, btcPairIndex, 49400));
    entries = vm.getRecordedLogs();
    (,,,,,,percentProfit,,) = abi.decode(entries[19].data, (uint, uint,
↪   ITradingStorage.Trade, ITradingStorage.LimitOrder, uint, uint, int, uint,
↪   bool));
    console.log("User 2 got his SL guaranteed");
    console2.log("User 2 percentProfit:", percentProfit);
}
```

Log output:

```
Logs:
  User decides to market close at ~ -5%
  Operator triggers market close at ~ -12%, even though user SL is at ~ -10%
  User 1, without slippage protection, experienced a funds loss of an extra 2% as
   ↪  his stop price was not triggered
  User 1 percentProfit: -121582328498
  User 2 got his SL guaranteed
  User 2 percentProfit: -101585531592
```

## Mitigation

Either respect the `guaranteed` stop loss in the `Close` market orders or introduce slippage protection to the `closeTradeMarket()` function.

# Issue M-13: User collateral value may be increased during a `Trading.updateMargin()` call without any USDC transfer in `useBackupOnly` mode.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/145

## Found by

Ironsidesec, TurnipBoy, eeyore, jah, jokr, samuraii77

## Summary

If the Governor switches the system to `useBackupOnly`, a malicious user can exploit this condition by performing a `Trading.updateMargin()` `DEPOSIT` action, effectively increasing their collateral without transferring the necessary USDC.

This occurs because `Trading.updateMargin()` is a user-triggered action that calls `PriceAggregator.fulfill()`. In `useBackupOnly` mode, the price sent to `TradingCallbacks.updateMarginCallback()` will always be 0. Although `updateMarginCallback()` verifies if `price!=0`, it does not revert if the price is 0. As a result, the entire `if` block is skipped, including the users required USDC transfer.

Furthermore, the system updates the users open trade data with the new `DEPOSIT` amount, artificially increasing `initialPosToken` without any USDC backing. Consequently, the system data becomes corrupted, making it impossible to exit `useBackupOnly`, liquidate the user position accurately, or distribute fees correctly, leading to potential protocol losses.

## Root Cause

Prematurely updating the user trade (here) and failing to revert when `price=0` (here) in `useBackupOnly` mode (here) enables collateral increases without any USDC deposit (here).

## Internal pre-conditions

1. The system is set to `useBackupOnly` state.

## External pre-conditions

None.

## Attack Path

1. The user calls `updateMargin()` to deposit collateral.
2. The users trade data is updated based on the deposit amount, with `initialPosToken` and `leverage` adjusted to reflect the original open interest.
3. In `useBackupOnly` mode, the `UPDATE_MARGIN` order type bypasses the `if/else` block in `fulfill()`, setting `price=0` for `updateMarginCallback()`.
4. Within `updateMarginCallback()`, the `if` block is skipped due to `price=0`.
5. The transaction completes without transferring USDC from the user to the Vault Manager.

## Impact

- Direct funds loss if `useBackupOnly` is lifted.
- Data corruption in storage.
- Incorrect fee accounting upon position liquidation.
- Potential protocol loss if the position is force-closed.

## PoC

Add the following to `IsolatedMargin.t.sol` and execute with `forgetest-vv--match-testtest_PocDeposit`:

```
function test_PocDeposit() public {
    vm.startPrank(traders[0]);
    usdc.transfer(traders[2], usdc.balanceOf(traders[0]));
    uint amount = 500e6;
    usdc.mint(traders[0], amount);
    usdc.approve(address(tradingStorage), amount);

    uint id = _placeMarketLong(traders[0], amount, btcPairIndex,  50000);
    vm.stopPrank();
    _executeMarketLong(traders[0], amount, btcPairIndex,  50000, id);
    console2.log("Position opened.");

    ITradingStorage.Trade memory _trade =
        tradingStorage.openTrades(traders[0], btcPairIndex, 0);
    console2.log("User USDC balance:", usdc.balanceOf(traders[0]));
    console2.log("User trade initialPosToken balance:", _trade.initialPosToken);

    vm.roll(1641070800);
    bytes[] memory priceUpdateData = _generateSampleUpdateDataCrypto(1,
↪   btcPairIndex, 50000);
```

```
    vm.startPrank(deployer);
    priceAggregator.useBackUpOracleOnly(true);
    vm.stopPrank();

    vm.startPrank(traders[0]);
    trading.updateMargin{value: mockPyth.getUpdateFee(priceUpdateData)}(
        btcPairIndex,
        0,
        ITradingStorage.updateType.DEPOSIT,
        amount,
        priceUpdateData);
    vm.stopPrank();
    console2.log("Margin updated.");

    ITradingStorage.Trade memory _updatedTrade =
        tradingStorage.openTrades(traders[0], btcPairIndex, 0);

    console2.log("User USDC balance:", usdc.balanceOf(traders[0]));
    console2.log("User trade initialPosToken balance:",
↪   _updatedTrade.initialPosToken);
}
```

Console log output:

```
Logs:
  Position opened.
  User USDC balance: 0
  User trade initialPosToken balance: 494000000
  Margin updated.
  User USDC balance: 0
  User trade initialPosToken balance: 993957665
```

As shown, the user inflated collateral by 500 USDC; `initialPosToken` increased without a USDC transfer.

## Mitigation

Revert in `updateMarginCallback()` when `price=0`.

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/39

# Issue M-14: Incorrect calculations in the Chainlink backup feed price deviation calculation.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/146

The protocol has acknowledged this issue.

## Found by

Ironsidesec, bughuntoor, eeyore, samuraii77

## Summary

An incorrect divisor is used in the deviation calculation when the Chainlink backup price is less than the Pyth price. This can result in unintended reverts even when the price is within the allowed deviation.

## Root Cause

The issue stems from using `bkPrice` as the divisor in the `if(bkPrice<price)` case (here), whereas the correct divisor should be `price`. This causes the deviation to be calculated from `bkPrice` instead of from the intended `price`.

## Internal pre-conditions

1. The pair in addition to Pyth is using a Chainlink backup feed.
2. The backup maxDeviation is set to 2% (or any other valid value).

## External pre-conditions

None.

## Attack Path

1. A transaction is executed for a pair that is using the Chainlink backup feed.
2. The Chainlink reported price is approximately -2% from the Pyth price.
3. The transaction is incorrectly reverted in a situation where it should be accepted.

## Impact

- Reverting time-sensitive function.
- Loss of fees, as the system may subsequently decide to `cancelPendingMarketOrder()`. In case it was a market order.

## PoC

Consider a simple calculation with a BTC price of 50,000 and a 2% backup deviation. This deviation allows an acceptable range of 49,000 to 51,000.

Currently, if Pyth reports a price of 50,000 and the Chainlink backup feed is 49,000, this should be within the acceptable deviation, and the transaction should proceed. However, due to the incorrect divisor, the transaction is reverted as follows:

```
(50000 - 49000) * 100 / 49000 > 2%

(50000 * 10**10 - 49000 * 10**10) * 100 * 10**10 / (49000 * 10**10) = 20408163265 >
↪  2%
```

Using the correct `price` divisor would yield the intended 2%:

```
(50000 * 10**10 - 49000 * 10**10) * 100 * 10**10 / (50000 * 10**10) = 20000000000
```

## Mitigation

Use `price` as the divisor in the `if(bkPrice<price)` case.

# Issue M-15: Incorrect `depositCap` check in `Tranche` contract.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/148

The protocol has acknowledged this issue.

## Found by

eeyore, samuraii77

## Summary

The `depositCap` check in the `_deposit` function of the `Tranche` contract uses an `assets` amount that includes a fee. However, this fee is transferred out of the vault, meaning it does not contribute to the actual asset amount in the vault and thus should not impact the `depositCap`.

## Root Cause

In the line `require(totalAssets()+assets<depositCap,"DEPOSIT_CAP_BREACHED");`, the function checks `assets` without accounting for the deduction of `getDepositFeesTotal(assets)`. As a result, the `depositCap` check is inflated by the fee amount, potentially causing deposits to be rejected incorrectly when the vault is close to the cap. The check should use `assets-fee` to accurately reflect the true impact on the vault balance. (here)

## Internal pre-conditions

1. The vault is close to reaching its `depositCap`.

## External pre-conditions

None.

## Attack Path

1. A user attempts to deposit when the vault is close to its `depositCap`.
2. Due to the fee being included in the `assets` amount, the cap check fails, rejecting the deposit unnecessarily.

## Impact

The current implementation may cause users to be incorrectly prevented from depositing due to an inflated `assets` value. This issue is more prominent when the vault balance is near the `depositCap`, as it could block further deposits prematurely.

## Mitigation

Modify the `depositCap` check to account for the deposit fee by using `assets-getDepositFeesTotal(assets)`, ensuring only the net deposited amount contributes to the cap calculation.

# Issue M-16: Incorrect closing fee calculation occurs because the `AccumulatedMarginFee` is not deducted from the `AdjustedPositionSize`, resulting in users paying higher fees.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/150

The protocol has acknowledged this issue.

## Found by

eeyore

## Summary

According to the Avantis documentation (doc), the closing fee should be based on the adjusted position size:

```
AdjustedPositionSize=TotalPositionSize+AccruedPnL-AccumulatedMarginFee
```

The documentation provides a clear example:

> ```
> ifatraderputsup$100ofcollateralata30xleverage,thenthetotalpositionsizewouldbe$3,000.Afterdeductingtheopeningfee(andassumingnochangeinthepriceoftheunderlyingasset),theleveragedpositionsizewithanaccumulatedmarginfeeonthepositionof$10is$(3000-10)=$2990.Hence,theclosingfeeis2990*0.08%=$2.392.
> ```

However, during closing fee calculation, the `AccumulatedMarginFee` (rollover fee) is not deducted as expected.

## Root Cause

In the `TradingCallbacks` contract, where `AdjustedPositionSize` is calculated during position closing, the `AccumulatedMarginFee` is not accounted for and does not reduce the position size (here and here).

This leads to users paying higher closing fees than they should.

## Internal pre-conditions

None.

## External pre-conditions

None.

## Attack Path

1. User closes their position.
2. Closing fees are overinflated.

## Impact

- Incorrect closing fee calculations result in users losing funds.
- Documentation discrepancy.

## Mitigation

Correctly reduce the `AdjustedPositionSize` by the accumulated rollover fee as described in the documentation.

# Issue M-17: Incorrect data is passed to the `TradingStorage.withinExposureLimits()` function.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/153

The protocol has acknowledged this issue.

## Found by

Afriaudit, eeyore

## Summary

When `TradingStorage.withinExposureLimits()` is called from the `TradingCallbacks` contract, the check is performed using an inaccurate `OpenInterest` value.

The value passed is derived from `positionSizeUSDC` and `leverage`, where `positionSizeUSDC` has not yet been reduced by the open fee.

If the open fee will be deducted from `positionSizeUSDC`, the actual `OpenInterest` used to update the OI storage values would be lower than the value used in the `TradingStorage.withinExposureLimits()` check.

This discrepancy leads to valid trades that would fit within the OI limits being rejected, resulting in potential loss of funds due to fees paid for operations such as initiating a market open trade (`msg.value`) and missing fees for the protocol.

## Root Cause

`TradingStorage.withinExposureLimits()` uses an `OpenInterest` value that has not been reduced by the potential open fee, causing the rejection of valid `Open` market or limit orders that would otherwise fit within OI limits after fee deduction.

The incorrect behavior can be observed here and here.

## Internal pre-conditions

1. The OI limits are nearly reached.

## External pre-conditions

None.

## Attack Path

1. A user creates a pending `Open` market order (not `isPnl`), calculating the OI limits to fit the largest possible OI trade.
2. The open fee is not correctly deducted when `TradingStorage.withinExposureLimits()` is used, causing the user pending market order to be rejected.
3. The user collateral is returned, but the fee is retained by the protocol, based on the assumption that `anattemptwasmade`.

## Impact

- Time-sensitive function is not executed correctly.
- Loss of user funds due to fees.
- Loss of fees for protocol and referrer.

## Mitigation

For pending `Open` market or limit orders that are not a new `isPnl` orders, precalculate the open position fee and use `positionSizeUSDC-fee` and `leverage` to determine the `OpenInterest` passed to `TradingStorage.withinExposureLimits()`.

# Issue M-18: The closing fee is not factored into the liquidation price calculation.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/167

The protocol has acknowledged this issue.

## Found by

KupiaSec, bughuntoor, eeyore, jokr

## Summary

Based on Avantis documentation (docs) and general logic, the closing fee should be included in the liquidation price calculation, just as the rollover fee is. The documentation specifies:

> `CollateralHealthRatio=(NetCollateral+PnL-accumulatedmarginfee-closingfee)/NetCollateral`

Omitting the closing fee deduction results in the `getTradeLiquidationPrice()` function calculating a liquidation price that is higher than it should be.

## Root Cause

The standard closing fee is not deducted in the `getTradeLiquidationPricePure()` function (here). This discrepancy with the documentation creates potential for inaccurate liquidation price ranges.

## Internal pre-conditions

None.

## External pre-conditions

None.

## Attack Path

This is a straightforward calculation error.

## Impact

- Incorrect distribution between PnL and standard fees.
- The tranches reserves and OI will be affected by the trade that should be already liquidated.

## Mitigation

Add the closing fee to the liquidation price calculation.

# Issue M-19: Wrong skew impact spread will be returned for a trader who is opening the first long position against the already existing short positions for a particular pair.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/170

## Found by

Varun_05

## Summary

Wrong skew impact spread will be returned in case when there is already a short position and a trader opens a long position i.e the first long position on the same pair.

## Root Cause

In the following line https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/PairInfos.sol#L346 The function incorrectly returns 0 when openInterestUSDCLong is zero for a trading pair.

## Internal pre-conditions

Na

## External pre-conditions

Initially a short position must be opened on a trading pair and then the first long position should be opened.

## Attack Path

1.Suppose there is initially some short positions opened on a trading pair i.e openInterestUSDCShort not equal to zero.

2. Now a trader opens a long position on this trading pair which is currently short skewed. Now as the pair is short skewed currently therefore better price should be offered to the current trader i.e getSkewImpactSpread function should return

negative value as the trader is trying to balance the long/short position ratio but currently 0 is returned which is incorrect.

3. If this scenario had been reversed i.e there had been initially long position and then a short position was opened then the getskewImpactSpread function would have worked completely fine by not returning 0 instead returning a negative value.

## Impact

The long trader are not offered better prices for reducing the short skew thus there is no incentive for the traders to balance the trading pair.

## PoC

*No response*

## Mitigation

Modify the getSkewImpactSpread function as follows

```
function getSkewImpactSpread(uint _pairIndex, bool _isBuy, uint _leveragePosition,
↪  bool isPnl) public view returns(int256 spread){

        if(isPnl) return 0; // NO Skew Impact spread for Pnl Based orders

        int intPrecision =  int(_PRECISION);
        int skewParam = pairsStorage.pairSkewImpactMultiplier(_pairIndex);

        uint openInterestUSDCLong = storageT.openInterestUSDC(_pairIndex, 0);
        uint openInterestUSDCShort = storageT.openInterestUSDC(_pairIndex, 1);
---         if(openInterestUSDCLong == 0) return 0;
+++    if(openInterestUSDCLong + openInterestUSDCShort == 0) return 0;
        uint skewPct = _isBuy
                        ? (1e4 * openInterestUSDCLong) / (openInterestUSDCLong +
↪  openInterestUSDCShort)
                        : (1e4 * openInterestUSDCShort) / (openInterestUSDCLong +
↪  openInterestUSDCShort);
        uint skewPctAfter = _isBuy
                        ? (1e4 * (openInterestUSDCLong  + _leveragePosition)) /
↪  (openInterestUSDCLong + openInterestUSDCShort + _leveragePosition)
                        : (1e4 * (openInterestUSDCShort + _leveragePosition)) /
↪  (openInterestUSDCLong + openInterestUSDCShort + _leveragePosition);
        int rawSpread = (ABDKMathQuadExt.expInt(skewPctAfter, 1e4, _PRECISION) -
↪  ABDKMathQuadExt.expInt(skewPct,1e4, _PRECISION) + ABDKMathQuadExt.expInt((1e4 -
↪  skewPctAfter), 1e4, _PRECISION) - ABDKMathQuadExt.expInt((1e4 - skewPct), 1e4,
↪  _PRECISION));         spread = (skewParam*rawSpread)/intPrecision;
        spread = (skewParam*rawSpread)/intPrecision;
```

```
    }
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/52

# Issue M-20: If a trader of market order or limit order is blacklisted for USDC token, reserved USDC tokens cannot be released forever

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/177

The protocol has acknowledged this issue.

## Found by

Ironsidesec, KupiaSec, eeyore, samuraii77

## Summary

The protocol uses the collateral token as USDC. If a trader of market order or limit order is blacklisted for USDC token, closing order is reverted. As a result, the order cannot be closed and the reserved USDC tokens cannot be released forever.

## Root Cause

When the order is closed, the protocol transfers the USDC tokens to trader in the VaultManager._sendUSDCToTrader.

```
function _sendUSDCToTrader(address _trader, uint _amount) internal {
    [...]
    require(storageT.usdc().transfer(_trader, _amount));
    [...]
}
```

If a trader who opens the order is USDC blacklisted, closing order will be reverted.

```
function _unregisterTrade(
    ITradingStorage.Trade memory _trade,
    int _percentProfit,
    uint _collateral,
    uint _feeAmountToken,
    uint _lpFeeToken,
    bool _isPnl
) private returns (uint usdcSentToTrader) {
```

```
    [...]
    storageT.vaultManager().sendUSDCToTrader(address(storageT), feeAfterRebate -
↪   referrerRebate - vaultAllocation);
```

## Internal pre-conditions

None

## External pre-conditions

1. None

## Attack Path

None

## Impact

If a trader of market order or limit order is blacklisted for USDC token, his order cannot be closed and the reserved USDC tokens cannot be released forever.

## PoC

None

## Mitigation

In the `_sendUSDCToTrader` function, track the amount of tokens to transfer to trader instead of direct transferring to trader. And, add the claim function that the traders claim tracked amount of USDC tokens.

# Issue M-21: The slippage percent is not updated during updating an open limit order

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/190

## Found by

Ironsidesec, KupiaSec, bughuntoor, pseudoArtist, samuraii77

## Summary

When updating open limit orders, `Trading.updateOpenLimitOrder()` accepts `_slippageP` parameter to update the slippage percent. But in the inner implementation `TradingStorage.updateOpenLimitOrder()` doesn't update the slippage percent.

## Root Cause

The slippage percent is given to the [`Trading.updateOpenLimitOrder()`], but it is not updated in the inner implementation.

The  supports updating the slippage percent. It accepts `_slippageP` as a function parameter and sends the parameter to `TradingStorage.updateOpenLimitOrder()`.

```
427: function updateOpenLimitOrder(
428:         uint _pairIndex,
429:         uint _index,
430:         uint _price,
431: @>      uint _slippageP,
432:         uint _tp,
433:         uint _sl
434:     ) external whenNotPaused {
442:         o.slippageP = _slippageP;
446:         storageT.updateOpenLimitOrder(o);

449:     }
```

But in the implementation of , there is no code to set the slippage percent.

```
function updateOpenLimitOrder(OpenLimitOrder calldata _o) external override
↪   onlyTrading {
    if (!hasOpenLimitOrder(_o.trader, _o.pairIndex, _o.index)) {
        return;
    }
    OpenLimitOrder storage o =
↪   openLimitOrders[openLimitOrderIds[_o.trader][_o.pairIndex][_o.index]];
```

```
    o.positionSize = _o.positionSize;
    o.buy = _o.buy;
    o.leverage = _o.leverage;
    o.tp = _o.tp;
    o.sl = _o.sl;
    o.price = _o.price;
    o.block = block.number;
}
```

## Internal pre-conditions

None

## External pre-conditions

None

## Attack Path

None

## Impact

The slippage percent is not updated even though traders try to update the slippage.

## PoC

None

## Mitigation

It is recommended to change the code as following:

```
    function updateOpenLimitOrder(OpenLimitOrder calldata _o) external override
↪   onlyTrading {
        if (!hasOpenLimitOrder(_o.trader, _o.pairIndex, _o.index)) {
            return;
        }
        OpenLimitOrder storage o =
↪   openLimitOrders[openLimitOrderIds[_o.trader][_o.pairIndex][_o.index]];
        o.positionSize = _o.positionSize;
        o.buy = _o.buy;
        o.leverage = _o.leverage;
        o.tp = _o.tp;
```

```
        o.sl = _o.sl;
        o.price = _o.price;
+       o.slippageP = _o.slippageP;
        o.block = block.number;
    }
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/40

# Issue M-22: In the `Tranche.sol`, there is no slippage check to deposit and withdraw

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/193

The protocol has acknowledged this issue.

## Found by

KupiaSec, pseudoArtist, samuraii77, thekmj

## Summary

In the `Tranche.sol`, there is no slippage check to deposit, withdraw, mint and redeem. Users should pay balancing fee and balancing fee is calculated using balance of junior and senior tranches. As a result, when users deposit or withdraw, actual paid balancing fee can be different from expected.

## Root Cause

The  function returns `balancingFee` or 0 according to the `getDynamicReserveRatio`

```
function getBalancingFee(address tranche, bool isDeposit, uint256 assets) external
↪  view override returns (uint256) {
    if ((getDynamicReserveRatio(tranche, isDeposit, assets) * 100) >
↪  balancingDeltaThreshold) {
        if ((tranche == address(junior) && isDeposit) || (tranche ==
↪  address(senior) && !isDeposit)) {
            return balancingFee;
        }
    }
    if ((getDynamicReserveRatio(tranche, isDeposit, assets) * 100) < 1e4 -
↪  balancingDeltaThreshold) {
        if ((tranche == address(senior) && isDeposit) || (tranche ==
↪  address(junior) && !isDeposit)) {
            return balancingFee;
        }
    }
    return 0;
}
```

The  calculates the reserve ratio using current balances of tranches.

```
function getDynamicReserveRatio(address tranche, bool isDeposit, uint256 assets)
↪  public view returns(uint256){
```

```
    IERC20 asset = IERC20(junior.asset());
    if (asset.balanceOf(address(senior)) == 0 && asset.balanceOf(address(junior))
↪   == 0) {
        return targetReserveRatio;
    }
    if(tranche == address(junior)){
        return isDeposit
            ?    (100 * (asset.balanceOf(address(junior)) + assets)) /
                (asset.balanceOf(address(junior)) +
↪   asset.balanceOf(address(senior)) + assets)
            :    (100 * (asset.balanceOf(address(junior)) - assets)) /
                (asset.balanceOf(address(junior)) +
↪   asset.balanceOf(address(senior)) - assets);
    }
    else{
        return isDeposit
            ?    (100 * asset.balanceOf(address(junior))) /
                (asset.balanceOf(address(junior)) +
↪   asset.balanceOf(address(senior)) + assets)
            :    (100 * asset.balanceOf(address(junior))) /
                (asset.balanceOf(address(junior)) +
↪   asset.balanceOf(address(senior)) - assets);
    }

}
```

## Internal pre-conditions

None

## External pre-conditions

1. None

## Attack Path

Let's consider the following scenario:

- Alice tries to deposit and expected balancing fee is 0.
- Another users change the balance of junior and senior tranches before Alice's transaction is executed.
- Current balancing fee is changed to `balancingFee` and her transaction is executed. Alice should pay unexpected balancing fee and receive less shares than expected amount.

## Impact

Liquidity providers may receive fewer assets or shares than expected from the tranche when depositing or withdrawing.

## PoC

None

## Mitigation

Add the slippage check to the deposit, withdraw, mint and redeem function in the tranche contract.

# Issue M-23: Inconsistency `referrerRebate` fee allocation between opening and closing order

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/203

## Found by

Afriaudit, KupiaSec, Varun_05, eeyore, samuraii77

## Summary

When opening the order, `feeAfterRebate` is calculated not to contain `referrerRebate` and `vaultAllocation` is calculated using `feeAfterRebate`. However, when closing the order, `feeAfterRebate` is calculated to contain `referrerRebate`. This is inconsistent fee allocation.

## Root Cause

In the , `feeAfterRebate` does not contain `referrerRebate` and `vaultAllocation` is calculated using `feeAfterRebate`. And this is paid to vault for opening fee.

```
File: avantis-contracts\src\TradingStorage.sol
L622:   uint vaultAllocation = (feeAfterRebate * (100 - _callbacks.vaultFeeP())) /
↪  100;
        uint govFees = (feeAfterRebate * _callbacks.vaultFeeP()) / 100 >> 1;
        if (_usdc) IERC20(usdc).safeTransfer(address(vaultManager), vaultAllocation
↪  - referrerRebate);
        vaultManager.allocateRewards(vaultAllocation - referrerRebate, false);
        govFeesUSDC += govFees;
        devFeesUSDC += feeAfterRebate - vaultAllocation - govFees;
```

When closing trade market, `feeAfterRebate` contains `referrerRebate`. However, when closing trade market, `feeAfterRebate` does not contain `referrerRebate`.

In the , `feeAfterRebate` contains `referrerRebate` and `vaultAllocation` is calculated using `feeAfterRebate`. And this is paid to vault for closing fee.

```
File: avantis-contracts\src\TradingCallbacks.sol
L533:       uint vaultAllocation = ((feeAfterRebate - referrerRebate) * (100 -
↪  vaultFeeP)) / 100;
            uint govFees = (feeAfterRebate - referrerRebate - vaultAllocation) / 2;
            storageT.incrementClosingFees(
                feeAfterRebate - referrerRebate - vaultAllocation - govFees,
```

```
                govFees
            );
```

## Internal pre-conditions

None

## External pre-conditions

1. None

## Attack Path

None

## Impact

Fee allocation between opening and closing order is inconsistent

## PoC

None

## Mitigation

Make `referrerRebate` fee allocation between opening and closing order consistent.

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/46

# Issue M-24: The `VeTranche.getTotalLockPoints()` function results in a loss of precision

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/218

The protocol has acknowledged this issue.

## Found by

KupiaSec, eeyore

## Summary

The `getTotalLockPoints()` function does not return `totalLockPoints`; instead, it returns `totalLockPoints/_PRECISION`. This leads to a loss of precision when calculating rewards.

## Root Cause

As noted at line 115, the `getTotalLockPoints()` function returns `totalLockPoints/_PRECISION`, not `totalLockPoints`. This is a source of precision loss.

```
    function getTotalLockPoints() public view override returns (uint256) {
115     return totalLockPoints/_PRECISION;
    }
```

Let's consider the following scenario:

1. Lock points:
    - Alice: `10e12`
    - Bob: `8e12+9e5`
    - totalLockPoints: `18e12+9e5`
    - `getTotalLockPoints()` returns: `(18e12+9e5)/1e6=18e6`
2. `180e6` rewards are distributed.
    - `rewardsDistributedPerSharePerLockPoint=(180e6*1e18)/18e6=10e18`

```
    function _distributeRewards(uint256 rewards) internal returns (uint256) {
```

```
272     rewardsDistributedPerSharePerLockPoint += (rewards * (_PRECISION **3))
↪  / getTotalLockPoints();
        ...
```

3. Individual rewards:
    - Alice: `10e18*10e12/1e24=100e6`
    - Bob: `10e18*(8e12+9e5)/1e24=80e6+9`
    - Needed rewards: `100e6+80e6+9=180e6+9`

```
    function _updateReward(uint256 _id) internal {
        if(lastSharePoint[_id] == rewardsDistributedPerSharePerLockPoint )
↪   return;

298     uint256 pendingReward = ((rewardsDistributedPerSharePerLockPoint -
↪   lastSharePoint[_id]) *
                                tokensByTokenId[_id] *
                                lockMultiplierByTokenId[_id]) /
                                (_PRECISION **4);
        rewardsByTokenId[_id] += pendingReward;
        lastSharePoint[_id] =  rewardsDistributedPerSharePerLockPoint;
    }
```

As a result, the needed rewards exceed the actual total rewards.

## Internal pre-conditions

## External pre-conditions

## Attack Path

## Impact

The needed rewards may exceed the actual total rewards.

## PoC

## Mitigation

Use `totalLockPoints` instead of `totalLockPoints/_PRECISION`.

# Issue M-25: Fees is not sent to the operator in case of opening a limit order using the openTrade function

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/219

## Found by

KupiaSec, Varun_05, samuraii77

## Summary

Fees is incorrectly not sent to the operator in case of opening a limit order using the openTrade function.

## Root Cause

Currently sending the operator fees is used only when opening a market order or a market pnl order . Fees is send to the operator because it used to update the feed in the pyth contracts. In case of limit orders no fees is send to the operator even though the trader sends the fees with the function call. https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/Trading.sol#L318

## Internal pre-conditions

Na

## External pre-conditions

Na

## Attack Path

Following is openTrade function

```
function openTrade(
       ITradingStorage.Trade calldata t,
       IExecute.OpenLimitOrderType _type,
       uint _slippageP
   ) external payable whenNotPaused returns(uint orderId) {
```

```
        IPriceAggregator aggregator = storageT.priceAggregator();
        IPairStorage pairsStored = aggregator.pairsStorage();

        require(storageT.pendingOrderIdsCount(__msgSender()) <
↪   storageT.maxPendingMarketOrders(), "MAX_PENDING_ORDERS");

        require(
            storageT.openTradesCount(__msgSender(), t.pairIndex) +
                storageT.pendingMarketOpenCount(__msgSender(), t.pairIndex) +
                storageT.openLimitOrdersCount(__msgSender(), t.pairIndex) <
                storageT.maxTradesPerPair(),
            "MAX_TRADES_PER_PAIR"
        );

        require(t.positionSizeUSDC.mul(t.leverage) >=
↪   pairsStored.pairMinLevPosUSDC(t.pairIndex), "BELOW_MIN_POS");


        require(t.tp == 0 || (t.buy ? t.tp > t.openPrice : t.tp < t.openPrice),
↪   "WRONG_TP");
        require(t.sl == 0 || (t.buy ? t.sl < t.openPrice : t.sl > t.openPrice),
↪   "WRONG_SL");

        if (_type != IExecute.OpenLimitOrderType.MARKET && _type !=
↪   IExecute.OpenLimitOrderType.MARKET_PNL ) {

            require(
                t.leverage > 0 &&
                    t.leverage >= pairsStored.pairMinLeverage(t.pairIndex, false) &&
                    t.leverage <= pairsStored.pairMaxLeverage(t.pairIndex, false),
                "LEVERAGE_INCORRECT"
            );

            storageT.transferUSDC(__msgSender(), address(storageT),
↪   t.positionSizeUSDC);
            uint index = storageT.firstEmptyOpenLimitIndex(__msgSender(),
↪   t.pairIndex);

            storageT.storeOpenLimitOrder(
                ITradingStorage.OpenLimitOrder(
                    __msgSender(),
                    t.pairIndex,
                    index,
                    t.positionSizeUSDC,
                    t.buy,
                    t.leverage,
                    t.tp,
                    t.sl,
                    t.openPrice,
                    _slippageP,
```

```
                block.number,
                0
            )
        );

        aggregator.executions().setOpenLimitOrderType(__msgSender(),
↪  t.pairIndex, index, _type);

        emit OpenLimitPlaced(__msgSender(), t.pairIndex, index, t.buy,
↪  t.openPrice, 0, _type, _slippageP, t.positionSizeUSDC);
    } else {

        (bool sent, ) = payable(operator).call{value: msg.value}("");
        require(sent, "EXECUTION_FEE_NOT_SENT");
        require(
            t.leverage > 0 &&
                t.leverage >= pairsStored.pairMinLeverage(t.pairIndex,  _type ==
↪  IExecute.OpenLimitOrderType.MARKET_PNL) &&
                t.leverage <= pairsStored.pairMaxLeverage(t.pairIndex,  _type ==
↪  IExecute.OpenLimitOrderType.MARKET_PNL),
            "LEVERAGE_INCORRECT"
        );

        storageT.transferUSDC(__msgSender(), address(storageT),
↪  t.positionSizeUSDC);

        orderId = _type == IExecute.OpenLimitOrderType.MARKET ?
                    aggregator.getPrice(t.pairIndex,
↪  IPriceAggregator.OrderType.MARKET_OPEN) :
                    aggregator.getPrice(t.pairIndex,
↪  IPriceAggregator.OrderType.MARKET_OPEN_PNL);


        storageT.storePendingMarketOrder(
            ITradingStorage.PendingMarketOrder(
                ITradingStorage.Trade(
                    __msgSender(),
                    t.pairIndex,
                    0,
                    0,
                    t.positionSizeUSDC,
                    0,
                    t.buy,
                    t.leverage,
                    t.tp,
                    t.sl,
                    0
                ),
                0,
                t.openPrice,
```

```
                    _slippageP
                ),
                orderId,
                true
            );

            emit MarketOrderInitiated(__msgSender(), t.pairIndex, true, orderId,
↪  block.timestamp);
        }
    }
```

As can be seen form above when order type is reversal or momentum then no fees is sent to the operator and a new limit order is opened. Now lets see that when execute limit order function is called what happens. Following is execute limit order function

```
function executeLimitOrder(
        ITradingStorage.LimitOrder _orderType,
        address _trader,
        uint _pairIndex,
        uint _index,
        bytes[] calldata priceUpdateData
    ) external payable whenNotPaused onlyOperator {

        IPairStorage pairsStored =
↪  IPriceAggregator(storageT.priceAggregator()).pairsStorage();
        if (_orderType == ITradingStorage.LimitOrder.OPEN) {

            require(storageT.hasOpenLimitOrder(_trader, _pairIndex, _index),
↪  "NO_LIMIT");

        } else {
            ITradingStorage.Trade memory t = storageT.openTrades(_trader,
↪  _pairIndex, _index);

            require(t.leverage > 0, "NO_TRADE");
            require(_orderType != ITradingStorage.LimitOrder.SL || t.sl > 0,
↪  "NO_SL");

            if (_orderType == ITradingStorage.LimitOrder.LIQ) {
                uint liqPrice =  pairInfos.getTradeLiquidationPrice(
                                    t.trader,
                                    t.pairIndex,
                                    t.index,
                                    t.openPrice,
                                    t.buy,
                                    t.initialPosToken,
                                    t.leverage
                                );
```

```
                require(t.sl == 0 || (t.buy ? liqPrice > t.sl : liqPrice < t.sl),
↪  "HAS_SL");
            }else{
                require(block.timestamp - t.timestamp >=
↪  pairsStored.openCloseThreshold(t.pairIndex, t.initialPosToken.mul(t.leverage)),
↪  "EARLY_CLOSE");
            }
        }

        IPriceAggregator aggregator = storageT.priceAggregator();
        IExecute executor = aggregator.executions();

        IExecute.TriggeredLimitId memory triggeredLimitId =
↪  IExecute.TriggeredLimitId(
            _trader,
            _pairIndex,
            _index,
            _orderType
        );

        bool isPnl = storageT.priceAggregator().pairsStorage().getPosType(_trader,
↪  _pairIndex, _index);

        uint orderId = aggregator.getPrice(
            _pairIndex,
            _orderType == ITradingStorage.LimitOrder.OPEN
                ? IPriceAggregator.OrderType.LIMIT_OPEN
                : isPnl
                    ? IPriceAggregator.OrderType.LIMIT_CLOSE_PNL
                    :  IPriceAggregator.OrderType.LIMIT_CLOSE
        );

        storageT.storePendingLimitOrder(
            ITradingStorage.PendingLimitOrder(_trader, _pairIndex, _index,
↪  _orderType),
            orderId
        );

        executor.storeFirstToTrigger(triggeredLimitId, __msgSender());
        emit LimitOrderInitiated(_trader, _pairIndex, orderId, block.timestamp);

        aggregator.fulfill{value: msg.value}(orderId, priceUpdateData);
    }
```

As can be seen from above that the operator needs to send some eth along this function call in order to call fulfill function on aggregator as it updates the pyth price feed therefore while opening limit orders fees should be sent to the operator in case of reversal and momentum orders too.

## Impact

Operator is not paid fees in case of opening limit orders of type reversal or momentum.

## PoC

*No response*

## Mitigation

Modify the opentrade function as follows

```
function openTrade(
        ITradingStorage.Trade calldata t,
        IExecute.OpenLimitOrderType _type,
        uint _slippageP
    ) external payable whenNotPaused returns(uint orderId) {

        IPriceAggregator aggregator = storageT.priceAggregator();
        IPairStorage pairsStored = aggregator.pairsStorage();

        require(storageT.pendingOrderIdsCount(__msgSender()) <
↪   storageT.maxPendingMarketOrders(), "MAX_PENDING_ORDERS");

        require(
            storageT.openTradesCount(__msgSender(), t.pairIndex) +
                storageT.pendingMarketOpenCount(__msgSender(), t.pairIndex) +
                storageT.openLimitOrdersCount(__msgSender(), t.pairIndex) <
                storageT.maxTradesPerPair(),
            "MAX_TRADES_PER_PAIR"
        );

        require(t.positionSizeUSDC.mul(t.leverage) >=
↪   pairsStored.pairMinLevPosUSDC(t.pairIndex), "BELOW_MIN_POS");


        require(t.tp == 0 || (t.buy ? t.tp > t.openPrice : t.tp < t.openPrice),
↪   "WRONG_TP");
        require(t.sl == 0 || (t.buy ? t.sl < t.openPrice : t.sl > t.openPrice),
↪   "WRONG_SL");
  ++++      (bool sent, ) = payable(operator).call{value: msg.value}("");
   ++++   require(sent, "EXECUTION_FEE_NOT_SENT");

        if (_type != IExecute.OpenLimitOrderType.MARKET && _type !=
↪   IExecute.OpenLimitOrderType.MARKET_PNL ) {

            require(
                t.leverage > 0 &&
```

```
                t.leverage >= pairsStored.pairMinLeverage(t.pairIndex, false) &&
                t.leverage <= pairsStored.pairMaxLeverage(t.pairIndex, false),
            "LEVERAGE_INCORRECT"
        );

        storageT.transferUSDC(__msgSender(), address(storageT),
↪  t.positionSizeUSDC);
        uint index = storageT.firstEmptyOpenLimitIndex(__msgSender(),
↪  t.pairIndex);

        storageT.storeOpenLimitOrder(
            ITradingStorage.OpenLimitOrder(
                __msgSender(),
                t.pairIndex,
                index,
                t.positionSizeUSDC,
                t.buy,
                t.leverage,
                t.tp,
                t.sl,
                t.openPrice,
                _slippageP,
                block.number,
                0
            )
        );

        aggregator.executions().setOpenLimitOrderType(__msgSender(),
↪  t.pairIndex, index, _type);

        emit OpenLimitPlaced(__msgSender(), t.pairIndex, index, t.buy,
↪  t.openPrice, 0, _type, _slippageP, t.positionSizeUSDC);
    } else {

-------          (bool sent, ) = payable(operator).call{value: msg.value}("");
 ------         require(sent, "EXECUTION_FEE_NOT_SENT");
        require(
            t.leverage > 0 &&
                t.leverage >= pairsStored.pairMinLeverage(t.pairIndex,  _type
↪  == IExecute.OpenLimitOrderType.MARKET_PNL) &&
                t.leverage <= pairsStored.pairMaxLeverage(t.pairIndex,  _type
↪  == IExecute.OpenLimitOrderType.MARKET_PNL),
            "LEVERAGE_INCORRECT"
        );

        storageT.transferUSDC(__msgSender(), address(storageT),
↪  t.positionSizeUSDC);

        orderId = _type == IExecute.OpenLimitOrderType.MARKET ?
```

```
                    aggregator.getPrice(t.pairIndex,
↪  IPriceAggregator.OrderType.MARKET_OPEN) :
                    aggregator.getPrice(t.pairIndex,
↪  IPriceAggregator.OrderType.MARKET_OPEN_PNL);


            storageT.storePendingMarketOrder(
                ITradingStorage.PendingMarketOrder(
                    ITradingStorage.Trade(
                        __msgSender(),
                        t.pairIndex,
                        0,
                        0,
                        t.positionSizeUSDC,
                        0,
                        t.buy,
                        t.leverage,
                        t.tp,
                        t.sl,
                        0
                    ),
                    0,
                    t.openPrice,
                    _slippageP
                ),
                orderId,
                true
            );

            emit MarketOrderInitiated(__msgSender(), t.pairIndex, true, orderId,
↪  block.timestamp);
        }
    }
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/41

# Issue M-26: User fund loss and incorrect fee calculation during partial trade market close in an edge case scenario.

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/226

## Found by

eeyore, jokr, qpzm

## Summary

The `Trading.closeTradeMarket()` function allows for partial position closing. However, when partial closing occurs, an issue within the `TradingCallbacks._unregisterTrade()` function leads to user fund losses and incorrect fee calculations.

When the condition `if(_trade.initialPosToken==_collateral||(_collateral+totalFees>=_trade.initialPosToken))` evaluates to true solely due to the second part `(_collateral+totalFees>=_trade.initialPosToken)`, the position is closed as if the entire remaining collateral was requested for withdrawal (`_collateral=_trade.initialPosToken;`).

This condition causes direct user fund losses, as the total fees were already deducted from the users collateral during `pairInfos.getTradeValue()` calculations and reflected in the withdrawal amount `usdcSentToTrader`. When the condition `(_collateral+totalFees>=_trade.initialPosToken)` is met, the user is penalized again with his remaining collateral.

In addition to this double penalization, the funds are locked in the Vault Manager, and final fee calculations are made based on an incorrect value. The `_collateral` should reflect the remaining `_trade.initialPosToken` when calculating fees for closing the position.

## Root Cause

The broken condition in `if(_trade.initialPosToken==_collateral||(_collateral+totalFees>=_trade.initialPosToken))` causes direct user fund losses and incorrect fee accounting when `(_collateral+totalFees>=_trade.initialPosToken)` is met. (here)

## Internal pre-conditions

None.

## External pre-conditions

None.

## Attack Path

1. A user performs a partial close on an `OpenTrade`.
2. The `if/else` block in the `(_collateral+totalFees>=_trade.initialPosToken)` check matches, with `_collateral` less than `_trade.initialPosToken`.
3. User is penalized.
4. The `_collateral` used in fee calculations in `pairInfos.getTradeValue()` was incorrect.

## Impact

- Direct user fund losses.
- Fees calculated based on an incorrect `_collateral` value are undercalculated.

## PoC

Add the following to `MarketTrade.t.sol` and call `forgetest-vv--match-testtest_PocPartialClose`:

```
function test_PocPartialClose() public {
    vm.startPrank(traders[0]);
    usdc.transfer(traders[2], usdc.balanceOf(traders[0]));
    uint amount = 10000e6;
    usdc.mint(traders[0], amount);
    usdc.approve(address(tradingStorage), amount);

    uint id1 = _placeMarketLong(traders[0], amount / 2, btcPairIndex, 50000);
    uint id2 = _placeMarketLong(traders[0], amount / 2, btcPairIndex, 50000);
    vm.stopPrank();
    _executeMarketLong(traders[0], amount / 2, btcPairIndex, 50000, id1);
    _executeMarketLong(traders[0], amount / 2, btcPairIndex, 50000, id2);
    vm.startPrank(traders[0]);

    uint pnlRewardsBefore = vaultManager.pnlRewards();
    uint totalRewardsBefore = vaultManager.totalRewards();

    vm.warp(100000);

    ITradingStorage.Trade memory _trade =
        tradingStorage.openTrades(traders[0], btcPairIndex, 0);
```

```
    _setChainlinkBTC(50000);
    uint closed1 = _placeMarketClose(btcPairIndex, _trade.initialPosToken, 0,
↪  50000);
    uint closed2 = _placeMarketClose(btcPairIndex, 4900000000, 1, 50000);
    vm.stopPrank();
    _executeMarketClose(btcPairIndex, _trade.initialPosToken, 0, 50000, closed1);
    uint pnlRewardsAfter1 = vaultManager.pnlRewards() - pnlRewardsBefore;
    uint totalRewardsAfter1 = vaultManager.totalRewards() - totalRewardsBefore;
    uint userBalanceAfter1 = usdc.balanceOf(traders[0]);

    _executeMarketClose(btcPairIndex, _trade.initialPosToken, 0, 50000, closed2);
    uint pnlRewardsAfter2 = vaultManager.pnlRewards() - pnlRewardsBefore;
    uint totalRewardsAfter2 = vaultManager.totalRewards() - totalRewardsBefore;
    uint userBalanceAfter2 = usdc.balanceOf(traders[0]);

    assert(tradingStorage.openTrades(traders[0], btcPairIndex, 0).leverage == 0);
    assert(tradingStorage.openTrades(traders[0], btcPairIndex, 1).leverage == 0);

    console2.log("User loss:", userBalanceAfter1 - (userBalanceAfter2 -
↪  userBalanceAfter1));
    console2.log("Pnl reward loss:", pnlRewardsAfter1 - (pnlRewardsAfter2 -
↪  pnlRewardsAfter1));
    console2.log("Reward loss:", totalRewardsAfter1 - (totalRewardsAfter2 -
↪  totalRewardsAfter1));
}
```

Console output:

```
Logs:
  User loss: 39313147
  Pnl reward loss: 287140
  Reward loss: 319770
```

The log shows a direct loss of approximately $40 for the user when closing the position partially, compared to closing it fully, highlighting the edge case error.

## Mitigation

Remove `(_collateral+totalFees>=_trade.initialPosToken)` from the `if()` condition, as `totalFees` were already correctly deducted from the user's collateral.

```
- if (_trade.initialPosToken == _collateral || (_collateral + totalFees >=
↪  _trade.initialPosToken)){
+ if (_trade.initialPosToken == _collateral) {
        storageT.unregisterTrade(_trade.trader, _trade.pairIndex, _trade.index);
        pairInfos.resetTradeInitialAccess(_trade.trader, _trade.pairIndex,
↪  _trade.index);
        _collateral = _trade.initialPosToken;
```

```
    }
    else {
        storageT.registerPartialTrade(_trade.trader, _trade.pairIndex,
↪   _trade.index, _collateral);
    }
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/42

# Issue M-27: Incorrect Spread Percentage Used for PnL-Based Orders in fulfill Function of PriceAggregator Contract

Source: https://github.com/sherlock-audit/2024-09-avantis-judging/issues/230

## Found by

Afriaudit, Ironsidesec, KupiaSec, TurnipBoy, Varun_05, jokr, qpzm

## Summary

In the fulfill function of the PriceAggregator contract, the spread percentage (spreadP) for orders of type MARKET_OPEN_PNL is incorrectly derived. Instead of fetching the spread for PnL-based orders, the function defaults to using the spread percentage for regular orders.

## Root Cause

https://github.com/sherlock-audit/2024-09-avantis/blob/main/avantis-contracts/src/PriceAggregator.sol#L185

The issue lies in the fulfill function where spreadP is set using `pairsStorage.pairSpreadP(r.pairIndex,false)`, regardless of the OrderType. When OrderType.MARKET_OPEN_PNL is used, the PnL-based spread should instead be retrieved with pairsStorage.pairSpreadP(r.pairIndex, true).

```
if (answers.length > 0) {
    ICallbacks.AggregatorAnswer memory a = ICallbacks.AggregatorAnswer(
        orderId,
        _median(answers),
        pairsStorage.pairSpreadP(r.pairIndex, false)  // Issue: Not distinguishing
↪  between PnL-based orders
    );
```

## Internal pre-conditions

*No response*

## External pre-conditions

*No response*

## Attack Path

*No response*

## Impact

improper spread percentage usage for pnl Order type

## PoC

*No response*

## Mitigation

Update the fulfill function to check the OrderType before determining the spread percentage

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Avantis-Labs/avantis-contracts/pull/43

# Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.