Zellic

Prepared for
Brank Dev
Avantis Labs, Inc.

Prepared by
Nipun Gupta
Kuilin Li
Zellic

December 1, 2023

Avantis

Smart Contract Security Assessment

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue ↗, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io ↗ or follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io ↗.

1. Executive Summary

Zellic conducted a security assessment for Avantis Labs, Inc. from November 6th to November 24th, 2023. During this engagement, Zellic reviewed Avantis's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- How do the liquidations work? Are the liquidations triggered at correct prices?
- Are the stop-loss and take-profit orders triggered at correct prices?
- Is it possible for the protocol to incur bad debt?
- Does the protocol stay solvent during various scenarios?
- Will the oracle provide the correct price feeds? If not, what effects would that have?
- Is it possible to create trades that exploit a vulnerability in the protocol and extract value from the protocol?

1.2. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Bots responsible for tracking and executing liquidations, limit order, and stop-limit orders
- Bots responsible for unlocking overdue locked tranches, distributing rewards at regular intervals, setting orderbook depth for dynamic spread on crypto pairs, and snapshotting the current open PNL of all trades to update the buffer ratio

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

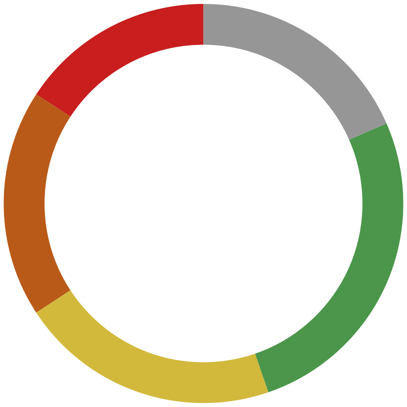
1.3. Results

During our assessment on the scoped Avantis contracts, we discovered 38 findings. Six critical issues were found. Seven were of high impact, eight were of medium impact, 10 were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Avantis Labs, Inc.'s benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	6
🟧 High	7
🟨 Medium	8
🟩 Low	10
◼ Informational	7



2. Introduction

2.1. About Avantis

Avantis is developing a user-friendly, decentralized, leveraged trading platform where users can long or short synthetic crypto, FOREX, and commodities using a financial primitive called perpetuals.

Synthetic leverage combined with a USDC stablecoin LP makes Avantis very capital efficient, allowing for a wide selection of tradable assets and high leverage (up to 100x). They are also unlocking fine-grained risk management for LPs via time and risk parameters, allowing any LP to be a sophisticated market maker for all kinds of derivatives, starting with perpetual.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Avantis Contracts

Repository	https://github.com/brankdev/avantis-contracts ↗
Version	avantis-contracts: `6f5c88ced715c2346dd805f2a93782105fd49254`
Programs	• Execute • PairInfos • PairStorage • PriceAggregator • Referral • Trading • TradingCallbacks • TradingStorage • Tranche • VaultManager • VeTranche
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io ↗

Kuilin Li
Engineer
kuilin@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 3, 2023	Kick-off call
November 6, 2023	Start of primary review period
November 24, 2023	End of primary review period

3. Detailed Findings

3.1. Stop loss higher than `openPrice` may cause fund loss

Target	TradingCallbacks		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The function `openTrade` is used to open a new market or limit trade. While creating a limit order, the `openPrice` could be set to any value. For a buy order, the value of stop loss should always be less than this `openPrice` due to the check `t.sl < t.openPrice` as shown below in `openTrade`:

```
function openTrade(
    ITradingStorage.Trade calldata t,
    IExecute.OpenLimitOrderType _type,
    uint _slippageP,
    bytes[] calldata priceUpdateData,
    uint _executionFee // In USDC. Optional for Limit orders
) external payable onlyWhitelist whenNotPaused {
    //...
    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");

    storageT.transferUSDC(msg.sender, address(storageT),
        t.positionSizeUSDC + _executionFee);

    if (_type != IExecute.OpenLimitOrderType.MARKET) {
        uint index = storageT.firstEmptyOpenLimitIndex(msg.sender,
            t.pairIndex);

        storageT.storeOpenLimitOrder(
            ITradingStorage.OpenLimitOrder(
                msg.sender,
                t.pairIndex,
                index,
                t.positionSizeUSDC,
                t.buy,
                t.leverage,
                t.tp,
                t.sl,
        t.openPrice,
        t.openPrice,
        block.number,
        _executionFee
      )
    );

    aggregator.executions().setOpenLimitOrderType(msg.sender,
      t.pairIndex, index, _type);

    emit OpenLimitPlaced(msg.sender, t.pairIndex, index,
      block.timestamp, _executionFee);
  } else //...
}
```

When a limit order is created, the `maxPrice` and the `minPrice` are set to the `openPrice` provided by user. If the order type is `MOMENTUM`, the `executeLimitOrder` could be called without errors as the require check `a.price <= o.maxPrice` in the `executeLimitOpenOrderCallback` callback function would pass as shown below:

```
function executeLimitOpenOrderCallback(AggregatorAnswer memory a)
external override onlyPriceAggregator {
  //...
  if (pairsStored.pairGroupIndex(o.pairIndex) == 0) {
    // crypto only
    (, uint priceAfterImpact) = pairInfos.getTradePriceImpact(
      _marketExecutionPrice(a.price, a.spreadP, o.buy),
      o.pairIndex,
      o.buy,
      o.positionSize.mul(o.leverage)
    );

    a.price = priceAfterImpact;
  } else {
    a.price = _marketExecutionPrice(a.price, a.spreadP, o.buy);
  }

  if (
   t == IExecute.OpenLimitOrderType.MARKET
    ? (a.price >= o.minPrice && a.price <= o.maxPrice)
    : (
     t == IExecute.OpenLimitOrderType.REVERSAL
      ? (o.buy ? a.price >= o.maxPrice : a.price <= o.minPrice)
      : (o.buy ? a.price <= o.maxPrice : a.price >= o.minPrice)
    ) && _withinExposureLimits(o.trader, o.pairIndex,
      o.positionSize.mul(o.leverage))
    ) {
        ITradingStorage.Trade memory finalTrade = _registerTrade(
            ITradingStorage.Trade(
                o.trader,
                o.pairIndex,
                0,
                0,
                o.positionSize,
                a.price, // current price
                o.buy,
                o.leverage,
                o.tp,
                o.sl, // large value
                0
            )
        );
```

Here, the `a.price` is the current price of the token, `o.sl` is still larger than this price, and the trade would be registered. As the stop loss is greater than the current price of the token, the stop loss could be triggered successfully. When the stop loss is triggered, the `profitP` is calculated in the callback function `executeLimitCloseOrderCallback`:

```
function executeLimitCloseOrderCallback(AggregatorAnswer memory a)
external override onlyPriceAggregator {
    //...
    v.price = aggregator.pairsStorage().guaranteedSlEnabled(t.pairIndex)
      ? o.orderType == ITradingStorage.LimitOrder.TP ? t.tp :
        o.orderType == ITradingStorage.LimitOrder.SL
          ? t.sl
          : a.price
      : a.price;

    v.profitP = _currentPercentProfit(t.openPrice, v.price, t.buy,
        t.leverage);

    v.posToken = t.initialPosToken;
    v.posUSDC = t.initialPosToken;

    if (o.orderType == ITradingStorage.LimitOrder.LIQ) {
        uint liqPrice = pairInfos.getTradeLiquidationPrice(
            t.trader,
            t.pairIndex,
            t.index,
            t.openPrice,
            t.buy,
            v.posUSDC,
            t.leverage
        );
        v.reward = (t.buy ? a.price <= liqPrice : a.price >= liqPrice) ?
            (v.posToken * liqFeeP) / 100 : 0;
    } else {
        v.reward = (o.orderType == ITradingStorage.LimitOrder.TP &&
            t.tp > 0 &&
            (t.buy ? a.price >= t.tp : a.price <= t.tp)) ||
            (o.orderType == ITradingStorage.LimitOrder.SL &&
                t.sl > 0 &&
                (t.buy ? a.price <= t.sl : a.price >= t.sl))
            ? (
                v.posToken.mul(t.leverage) *
                aggregator.pairsStorage().pairLimitOrderFeeP(t.pairIndex)
            ) / 100 / _PRECISION
            : 0;
    }

    if (o.orderType == ITradingStorage.LimitOrder.LIQ && v.reward > 0) {
        uint usdcSentToTrader = _unregisterTrade(
            t,
            v.profitP,
            v.posUSDC,
            v.reward,
            (v.reward * (liqTotalFeeP - liqFeeP)) / liqFeeP,
            i.lossProtection
        );
```

When the `profitP` is calculated, it will set the `v.price` to the `t.sl`, which was higher than the current price of the token. Due to the vulnerability — as the `t.sl` could be set higher than the `t.openPrice` — it would always return in positive net profit when this function is called. The stop loss could be set to a really large value, leading to the maximum profit (900%) as allowed by the protocol and fund loss.

Here is an example trade:

```
Start
  Balance of trader before: 10000000000
  Trader places a limit order with large openPrice and stop loss just below
    it. Here is an example trade:
    trade.trader = _trader;
    trade.pairIndex = _pairIndex;
    trade.index = _index;
    trade.initialPosToken = 0;
    trade.positionSizeUSDC = _amount;
    trade.openPrice =  100000e10;
```

```
     trade.buy = true;
     trade.leverage = 10e10;
     trade.tp = 0;
     trade.sl = 100000e10-1;
     trade.timestamp = block.number;

    When the open order is executed, the openPrice is changed to the current
     price + price impact:
     OpenPrice for the long order 505252500000000
     Stop loss for the long order 999999999999999

    Stop loss is then immediately executed as the value of stop loss is
     greater than the current price.
    Balance of trader after:  97911000000
```

Impact

An attacker can create a malicious trade such that they always make 900% returns instantly. They could use this to drain the protocol.

Recommendations

After we discussed the issue with the Avantis team, they suggested removing the `MOMENTUM` order type, which prevents users from setting stop loss higher than the current price and prevents this issue.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 2609b4c5 ↗.

3.2. Unsafe cast in take profit can lead to fund loss

Target	TradingCallbacks		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

For any open trade, the value of take profit can be set to any large value using the functions updateTp and updateTpAndSl. If the value of _newTP is set to type(uint256).max for a sell order, the function _currentPercentProfit will calculate the percent profit as the max gain percent (900%) due to the casting from uint256 to int.

```
function _currentPercentProfit(
        uint openPrice,
        uint currentPrice,
        bool buy,
        uint leverage
    ) private pure returns (int p) {
        int diff = buy ? (int(currentPrice) - int(openPrice)) :
    (int(openPrice) - int(currentPrice));
        int minPnlP = int(_PRECISION) * (-100);
        int maxPnlP = int(_MAX_GAIN_P) * int(_PRECISION);
        p = (diff * 100 * int(_PRECISION.mul(leverage))) / int(openPrice);
        p = p < minPnlP ? minPnlP : p > maxPnlP ? maxPnlP : p;
    }
```

For example, the TP can be updated to type(uint256).max using updateTp or updateTpAndSl. In case executeLimitOrder is called of the type LimitOrder.TP for a sell order, the value of currentPrice in _currentPercentProfit will be type(uint256).max. When this value is casted to int, it will become -1, and thus the diff would be openPrice + 1 and p will become ≈100*int(_PRECISION.mul(leverage)). Therefore, if the leverage is greater than nine, the function will return the profit as 900%.

The value of currentPrice can be set to t.tp in executeLimitCloseOrderCallback if the trade is of type LimitOrder.TP as shown below.

```
v.price = aggregator.pairsStorage().guaranteedSlEnabled(t.pairIndex)
? o.orderType == ITradingStorage.LimitOrder.TP ? t.tp : o.orderType ==
    ITradingStorage.LimitOrder.SL
    ? t.sl
    : a.price
: a.price;

v.profitP = _currentPercentProfit(t.openPrice, v.price, t.buy, t.leverage);
```

Here is an example trade to execute the attack:

```
Start
  Balance of trader before: 1000000000
  Trader creates a short position:
    trade.trader = _trader;
    trade.pairIndex = _pairIndex;
    trade.index = _index;
    trade.initialPosToken = 0;
    trade.positionSizeUSDC = _amount;
    trade.openPrice =  _price;
    trade.buy = true;
    trade.leverage = 10e10;
    trade.tp = 0;
    trade.sl = 0;
    trade.timestamp = block.number;

  Trader calls `updateTp` with the `_newTp` as
    `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff`

  When execute limit close is called of type `LimitOrder.TP`, the trader
    receives 900% profit.

  Balance of trader after: 9791100000
```

Impact

An attacker can create a malicious trade such that they always make 900% returns instantly. They could use this to drain the protocol.

Recommendations

While setting a new TP price, use the function `_correctTp` to check if the TP is in correct range using the callback.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `de74d560` ↗.

3.3. The function `setWithdrawThreshold` lacks access control

Target	Tranche		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The external function `setWithdrawThreshold` is used to allow governance to set the withdraw threshold parameter:

```
function setWithdrawThreshold(uint256 _withdrawThreshold) external {
    require(_withdrawThreshold < 100 * _PRECISION, "THRESHOLD_EXCEEDS_MAX");
    withdrawThreshold = _withdrawThreshold;
    emit WithdrawThresholdUpdated(_withdrawThreshold);
}
```

However, this function lacks all access control.

Impact

Anyone can update the withdraw threshold at any time.

Front-runners can cause user withdrawals to revert by setting the `withdrawThreshold` to zero. Users can change the `withdrawThreshold` to withdraw more than intended.

Recommendations

Add the missing `onlyGov` modifier to this function.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `4e5b1384` ↗.

3.4. Reserve requirement and fees checked before withdrawal

Target	Tranche		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When withdrawing from Tranche, the utilization ratio is checked in `_withdraw` so that the withdraw does not cause the protocol to become insolvent:

```
function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal virtual override {
    require(utilizationRatio() < withdrawThreshold,
        "UTILIZATION_RATIO_MAX");

    uint256 fee = getWithdrawalFeesRaw(assets);

    super._withdraw(caller, receiver, owner, assets, shares);
```

However, the call to `utilizationRatio()` calls `super.totalAssets()`, which checks the current assets of the contract, and that has not changed yet. Similarly, `getWithdrawalFeesRaw` uses the current assets in the contract, and only later in `super._withdraw` do the assets get transferred out.

This means that the utilization ratio and withdrawal fees are calculated on the prewithdrawal state, not the postwithdrawal state. The same thing happens with the balancing fee on deposit.

Impact

As long as the utilization ratio is currently not violated, a share owner can withdraw any amount, including an amount that would leave the utilization ratio violated after the USDC moves out of the Tranche. Here is a proof-of-concept (POC) output:

```
Start
```

```
 - Junior reserved = 0 actual = 0
 - Senior reserved = 0 actual = 0
LP provides liquidity
 - Junior reserved = 0 actual = 995024875621
 - Junior util ratio % = 0
 - Senior reserved = 0 actual = 1000000000000
 - Senior util ratio % = 0
Traders open market longs
 - Junior reserved = 92609441060 actual = 995024875621
 - Junior util ratio % = 9
 - Senior reserved = 49866622110 actual = 1000000000000
 - Senior util ratio % = 4
LP withdraws liquidity
 - Junior reserved = 92609441060 actual = 4777375621
 - Junior util ratio % = 1938
 - Senior reserved = 49866622110 actual = 9752500000
 - Senior util ratio % = 511
```

At the end of the POC, the utilization ratio for the tranches were 1938% and 511%, when it should never be above 100%. This means the protocol has become insolvent and cannot pay all trader profits or return trader collateral.

Additionally, even if the utilization ratio is currently violated, a share owner can still withdraw any amount by first depositing a flash loan that brings the utilization ratio back under the threshold and then withdrawing both the flash loan and the amount they wanted to withdraw.

The balancing fees on deposit and withdrawal fees on withdrawal can similarly be dodged with a flash loan.

Recommendations

Ensure all relevant quantities used in these checks are calculated on post-action balances.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 2c45e310 ↗.

3.5. Locked shares have undue access to historical rewards

Target	VeTranche		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In VeTranche, the `rewardsDistributedPerSharePerLockPoint` state variable increases in order to give rewards to all of the current locked positions. Each time rewards are distributed at the request of the owner of the lock or when the lock expires, the difference between the current value of this state variable and the last checkpoint for this variable is awarded:

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

    uint256 pendingReward =
        ((rewardsDistributedPerSharePerLockPoint - lastSharePoint[_id]) *
        tokensByTokenId[_id] *
        lockMultiplierByTokenId[_id]) /
        (_PRECISION **3);
    rewardsByTokenId[_id] += pendingReward;
    lastSharePoint[_id] = rewardsDistributedPerSharePerLockPoint;
}
```

However, when a locked position is initially created, `lastSharePoint[id]` is not set to the current value of `rewardsDistributedPerSharePerLockPoint`, and it is uninitialized and zero:

```
function lock(uint256 shares, uint endTime)
public nonReentrant returns (uint256) {
    //...
    tokensByTokenId[nextTokenId] = shares;
    lockTimeByTokenId[nextTokenId] = endTime;
    lockStartTimeByTokenId[nextTokenId] = block.timestamp;
    rewardsByTokenId[nextTokenId] = 0;
    lockMultiplierByTokenId[nextTokenId] =
        getLockPoints(endTime - block.timestamp);
    totalLockPoints +=
        (shares * lockMultiplierByTokenId[nextTokenId]) / _PRECISION;
```

Impact

A newly locked position will be incorrectly awarded a share of all the rewards distributed before it was locked. This causes the VeTranche to immediately become insolvent because the expected amount of USDC it holds is more than the amount it will actually hold. The impact is that later reward claimants will not be able to claim rewards.

Here is the output from the POC:

```
Start
LP 1 locks
LP 2 locks
 - rewardsDistributedPerSharePerLockPoint = 0
 - LP 1 reward = 0
 - LP 2 reward = 0
 - LP 3 reward = 0
Warp 7 days, rewards distributed
LP 1 unlocks
 - rewardsDistributedPerSharePerLockPoint = 811376886
 - LP 1 reward = 28815579507696
 - LP 2 reward = 0
 - LP 3 reward = 0
LP 3 locks and immediately unlocks
 - rewardsDistributedPerSharePerLockPoint = 811376886
 - LP 1 reward = 28815579507696
 - LP 2 reward = 0
 - LP 3 reward = 12738897024053
LP 2 tries to unlock
[FAIL. Reason: ERC20: transfer amount exceeds balance]
```

After rewards are distributed, LP 3 locks and then immediately unlocks, which means they should not get any rewards. However, it does get rewards, and then after this, LP 2 cannot unlock due to insufficient funds in the VeTranche.

Recommendations

Set the `lastSharePoint[id]` checkpoint to the current value of `rewardsDistributedPerSharePerLockPoint` in the `lock` function.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `fc4d4be1` ↗.

3.6. Max profit can exceed amount reserved from vault

Target	TradingCallbacks		
Category	Business Logic	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

When a new trade is finalized in `_registerTrade`, the amount reserved from the vault to ensure that the trade can be closed is the leveraged position size:

```
function _registerTrade(ITradingStorage.Trade memory _trade)
private returns (ITradingStorage.Trade memory) {
    //...
    storageT.vaultManager().reserveBalance(
        _trade.positionSizeUSDC.mul(_trade.leverage));
    storageT.vaultManager().receiveUSDCFromTrader(
        _trade.trader, _trade.positionSizeUSDC, 0);
```

This leveraged position size is calculated by multiplying the unleveraged position size `positionSizeUSDC` by the leverage.

However, the amount that the trader receives when closing the position is only capped at 900% of the unleveraged position size:

```
function _currentPercentProfit(
    uint openPrice,
    uint currentPrice,
    bool buy,
    uint leverage
) private pure returns (int p) {
    int diff = buy ? (int(currentPrice) - int(openPrice))
                   : (int(openPrice) - int(currentPrice));
    int minPnlP = int(_PRECISION) * (-100);
    int maxPnlP = int(_MAX_GAIN_P) * int(_PRECISION);
    p = (diff * 100 * int(_PRECISION.mul(leverage))) / int(openPrice);
    p = p < minPnlP ? minPnlP : p > maxPnlP ? maxPnlP : p;
}
```

Here, the constant `_MAX_GAIN_P` is 900%, and the function's return value is later multiplied by `positionSizeUSDC` to determine the amount of USDC sent back to the trader.

This means that if the leverage is less than 9x and the underlying price moves in the direction the trader expected, the amount reserved from the vault can be less than the amount the trader receives when they close the position.

Impact

The protocol can become insolvent if multiple low-leverage trades are profitable, because the amount reserved would be insufficient to cover rewards.

See this POC output, where an LP supplies liquidity, then several traders open long positions with 2x leverage, and then the underlying price increases 5x so the trades close at max profit:

```
Start
 - Junior reserved = 0 actual = 0
 - Senior reserved = 0 actual = 0
LP provides liquidity
 - Junior reserved = 0 actual = 995024875621
 - Junior util ratio % = 0
 - Senior reserved = 0 actual = 1000000000000
 - Senior util ratio % = 0
Traders open low-leverage market longs
 - Junior reserved = 18671560034 actual = 995024875621
 - Junior util ratio % = 1
 - Senior reserved = 10053916944 actual = 1000000000000
 - Senior util ratio % = 1
Traders close positions at max profit
 - Junior reserved = 0 actual = 920503530457
 - Junior util ratio % = 0
 - Senior reserved = 0 actual = 959873121832
 - Senior util ratio % = 0
```

Out of the junior tranche, the traders were in total awarded 995024875621 - 920503530457, which is about 7.5e10, but only 18671560034, which is about 1.8e10, was reserved. Similarly, about 4e10 was taken out of the senior tranche, but only about 1e10 was reserved.

The following output is the same scenario, except before the traders close their max-profit positions, the LP decides to remove most of the unreserved liquidity:

```
Start
 - Junior reserved = 0 actual = 0
 - Senior reserved = 0 actual = 0
LP provides liquidity
 - Junior reserved = 0 actual = 995024875621
 - Junior util ratio % = 0
 - Senior reserved = 0 actual = 1000000000000
 - Senior util ratio % = 0
Traders open low-leverage market longs
 - Junior reserved = 18671560034 actual = 995024875621
 - Junior util ratio % = 1
 - Senior reserved = 10053916944 actual = 1000000000000
 - Senior util ratio % = 1
LP withdraws most liquidity
 - Junior reserved = 18671560034 actual = 19361283100
 - Junior util ratio % = 96
 - Senior reserved = 10053916944 actual = 10309251944
 - Senior util ratio % = 97
Traders close positions at max profit
[FAIL. Reason: ERC20: transfer amount exceeds balance]
```

After the LP withdraws most of the liquidity, the utilization ratios are 96% and 97%, so even if Finding 3.4. ↗ is fixed, this withdrawal would still go through.

Next, the traders' calls to `Trading.closeTradeMarket` unfortunately revert because inside the `Tranche.withdrawAsVaultManager`, the call to `USDC.transfer` reverts as the tranches do not have enough funds to send out. This shows that the reserved quantity was not sufficient.

Recommendations

Instead of reserving the leveraged position size, reserve an amount equal to the max amount of USDC that the trader can potentially get back upon closing the position, maximized across all values the underlying price can be in the future. This quantity will depend on the max profit parameter and should not depend on the leverage.

Remediation

This issue has been acknowledged by Avantis as a risk that is within the risk tolerance of the protocol. The maximum open interest and other safety limits on individual trades are expected to collectively mitigate this risk.

3.7. Update margin uses new leverage for balance release

Target	TradingCallbacks		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The function `updateMargin` allows users to deposit or withdraw USDC from their open positions. When updating the margin order, there are fees (`marginFees`) that the users have to pay. These fees are allocated as rewards in the vault manager, and as the fees are taken from the origin order, it is essential to release that balance and update the open interest related to the fees.

The function `updateMargin` stores the pending margin-update order using the function `storePendingMarginUpdateOrder` and stores the new leverage in the struct's `oldLeverage` field. Later, in the callback function `updateMarginCallback`, the `o.marginFees.mul(o.oldLeverage)` value is used to release the balance and update the open interest.

As `o.oldLeverage` is the new leverage and not the old leverage, it would release an incorrect amount of balance, and the open interest would also be updated using this incorrect value.

Impact

If more tokens are released from the vault, in the long term it could lead to losing positions not being liquidated as there will not be enough balance in the vault to be released.

Also, using the new leverage and trade size makes the calculation for the withdrawal threshold check incorrect when it interacts with the -100% loss cap.

For example, say the original investment was $100 with leverage 100x, the loss-protection multiplier is 80%, and the asset price changes by -0.5%. Now, the position has a PNL of -$50, which is -$40 after the loss-protection multiplier, so the position is worth $60.

But, consider if the user tries to withdraw $75 minus a WEI, so the new position is $25 plus a WEI and the new leverage is slightly less than 400x. Now, because the leverage is higher, the percent profit will be -200%, reduced to -100% after the loss cap and then reduced to -80% due to the loss protection tier, so it is $20. The require statement checks if $25 + a WEI - $20 > ($25 + a WEI) * 80%, which is true, so it allows the user to withdraw the amount despite it being more than what the position is worth.

Recommendations

Pass the correct old leverage value in the `storePendingMarginUpdateOrder` function. Ensure that the callback correctly uses the old value when checking if the withdraw threshold is breached.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `d625039e` ↗.

3.8. Partial trades update open-interest incorrectly

Target	TradingStorage		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The function `registerPartialTrade` is used by `_unregisterTrade` to register a partial trade in case the collateral to be withdrawn is less than the initial position of the trade. When this partial trade is registered, the open interest is updated using the function `_updateOpenInterestUSDC` in `registerPartialTrade` as shown below:

```
function registerPartialTrade(
    address trader,
    uint pairIndex,
    uint index,
    uint _amountReduced
) external override onlyTrading {
    Trade storage t = _openTrades[trader][pairIndex][index];
    TradeInfo storage i = _openTradesInfo[trader][pairIndex][index];
    if (t.leverage == 0) {
        return;
    }
    t.initialPosToken -= _amountReduced;
    i.openInterestUSDC -= _amountReduced.mul(t.leverage);
    _updateOpenInterestUSDC(trader, pairIndex, i.openInterestUSDC,
        false, t.buy, t.openPrice);
}

function _updateOpenInterestUSDC(
    address _trader,
    uint _pairIndex,
    uint _leveragedPosUSDC,
    bool _open,
    bool _long,
    uint _price
) private {
    uint index = _long ? 0 : 1;
    uint[2] storage o = openInterestUSDC[_pairIndex];

    // Fix beacuse of Dust during partial close
    if (!_open) _leveragedPosUSDC =
        _leveragedPosUSDC > o[index] ? o[index] : _leveragedPosUSDC;

    o[index] = _open ?
        o[index] + _leveragedPosUSDC : o[index] - _leveragedPosUSDC;
    totalOI = _open ?
        totalOI + _leveragedPosUSDC : totalOI - _leveragedPosUSDC;
    _walletOI[_trader] = _open ?
        _walletOI[_trader] + _leveragedPosUSDC :
        _walletOI[_trader] - _leveragedPosUSDC;

    emit OIUpdated(_open, _long, _pairIndex, _leveragedPosUSDC, _price);
}
```

Here, the amount of open interest to be reduced should be equal to `_amountReduced.mul(t.leverage)`. But the argument passed to `_updateOpenInterestUSDC` is `i.openInterestUSDC`, which is the original open interest minus the new amount times leverage.

Impact

If the open interest is incorrectly updated, it would lead to incorrect returned values for the loss-protection tier, utilization multiplier, long multiplier, short multiplier, and rollover fees.

Recommendations

We recommend replacing `i.openInterestUSDC` with `_amountReduced.mul(t.leverage)` in the third parameter of the call to `_updateOpenInterestUSDC`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `cfb288e3` ↗.

3.9. Referrer rebates must not decrease `totalRewards`

Target	TradingCallbacks		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

When a trade is opened, if the trader has set their referral code, the referrer would get a rebate. This rebate is added as `rebates` in the function `applyReferralOpen` when a trade is opened. Additionally, part of the remaining fee, after the rebate, is allocated as rewards in the vault manager.

```
function handleDevGovFees(
    address _trader,
    uint _pairIndex,
    uint _leveragedPositionSize,
    bool _usdc,
    bool _fullFee,
    bool _buy
) external override onlyTrading returns (uint feeAfterRebate) {
    //...
    feeAfterRebate = applyReferralOpen(_trader, fee,
    _leveragedPositionSize);

    uint vaultAllocation =
        (feeAfterRebate * (100 - _callbacks.vaultFeeP())) / 100;
    uint govFees = (feeAfterRebate * _callbacks.vaultFeeP()) / 100 / 2;

    if (_usdc) usdc.transfer(address(vaultManager), vaultAllocation);

    vaultManager.allocateRewards(vaultAllocation);
    //...
}

function applyReferralOpen(
    address _trader,
    uint _fees,
    uint _leveragedPosition
) public override onlyTrading returns (uint) {
    (uint traderFeePostDiscount, address referrer, uint referrerRebate) =
        referral.traderReferralDiscount(_trader, _fees);
    if (referrer != address(0)) {
        rebates[referrer] += referrerRebate;
        emit TradeReferred(
            _trader,
            referrer,
            _leveragedPosition,
            traderFeePostDiscount,
            _fees - traderFeePostDiscount,
            referrerRebate
        );
        return traderFeePostDiscount - referrerRebate;
    }
    return _fees;
}
```

When the trade is closed, the function `applyReferralClose` returns the `referrerRebate`, which is then subtracted from `totalRewards` in the function `sendReferrerRebateToStorage` as shown:

```
function _unregisterTrade(
    ITradingStorage.Trade memory _trade,
    int _percentProfit,
    uint _collateral,
    uint _feeAmountToken, // executor reward
    uint _lpFeeToken,
    uint _tier
) private returns (uint usdcSentToTrader) {
    //Scoping Local Variables to avoid stack too deep
    uint totalFees;
    {
        (uint feeAfterRebate, uint referrerRebate) =
            storageT.applyReferralClose(
                _trade.trader,
                _lpFeeToken,
                _trade.initialPosToken.mul(_trade.leverage)
            );

        //...
        if (referrerRebate > 0) {
            storageT.vaultManager()
                .sendReferrerRebateToStorage(referrerRebate);
        }
    }
}
```

```
function sendReferrerRebateToStorage(uint _amount)
external override onlyCallbacks {
    require(_amount > 0, "NO_REWARDS_ALLOCATED");
    require(totalRewards >= _amount, "UNDERFLOW_DETECTED");

    totalRewards -= _amount;
    IERC20(junior.asset()).transfer(address(storageT), _amount);

    emit ReferralRebateAwarded(_amount);
}
```

As the rewards do not include the referral rebate amount, these should not be subtracted from `totalRewards`.

Impact

The `totalRewards` distributed will be less than the total rewards available in the vault manager.

Recommendations

We recommend not subtracting the referral rebate from `totalRewards`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 6e335dc2 ↗.

3.10. Precision loss in `totalLockPoints` causes undue rewards and insolvency

Target	VeTranche		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

In VeTranche, `totalLockPoints` counts the total share lockpoints of all the locked positions and is the denominator that divides all reward distributions to make sure all locks get the same quantity of reward per share per lockpoint. However, compared to the actual lockpoints, it has much less precision due to being divided by `_PRECISION`:

```
function lock(uint256 shares, uint endTime)
public nonReentrant returns (uint256) {
    //...
    lockMultiplierByTokenId[nextTokenId] =
        getLockPoints(endTime - block.timestamp);
    totalLockPoints +=
        (shares * lockMultiplierByTokenId[nextTokenId]) / _PRECISION;
```

This matters because the lock can be set for any number of shares and days, so the granularity with which the user has control over `lockMultiplierByTokenId` for a given token is much finer than `_PRECISION`:

```
function getLockPoints(uint256 timeLocked)
public view override returns (uint256) {
    uint256 lockedDays = timeLocked > getMinLockTime() ?
        (timeLocked - getMinLockTime()) / 86400 : 0;
    uint256 points = _PRECISION +
        (((lockedDays ** 2) * multiplierCoeff * _PRECISION)
            / multiplierDenom);
    return points;
}
```

For example, assuming the default parameters of `multiplierCoeff = 1815e5` and `multiplierDenom = 1960230 * _PRECISION`, if a user locks one share for exactly 103 days more than the minimum, then `points` will be about `1.9823 * _PRECISION`. After that is divided by `_PRECISION`, it rounds down to one.

Impact

Users wishing to lock shares can get more rewards than they are due if they tailor `shares` and `timeLocked` to take maximum benefit from this rounding.

In the most extreme case, if the user locks one share at a time for 103 days past the minimum amount of time, they get approximately twice the share than they are entitled to.

Extra USDC given out due to this precision loss causes reward insolvency, where the balance of VeTranche is less than the total outstanding rewards, which causes the last few unlockers to not be able to unlock because the VeTranche runs out of USDC.

See this POC output:

```
Start
 - totalLockPoints = 0
 - LP 1 reward = 0
 - LP 2 reward = 0
LP 1 locks 100 shares
 - totalLockPoints = 198
 - LP 1 reward = 0
 - LP 2 reward = 0
LP 2 locks 100x 1 share
 - totalLockPoints = 298
 - LP 1 reward = 0
 - LP 2 reward = 0
Distribute 1e10 rewards to VeTranche
LP 2 unlocks 100x
 - totalLockPoints = 198
 - LP 1 reward = 0
 - LP 2 reward = 6652010000
LP 1 tries to unlock
[FAIL. Reason: ERC20: transfer amount exceeds balance]
```

LP 1 and LP 2 both lock 100 shares in total, for the same amount of time, so they should both get half of the 1e10 WEI = $10,000 total reward distributed while they were locked, $5,000 each. However, when LP 1 locked their shares, `totalLockPoints` increased by 198, and when LP 2 locked their shares, since they did one share at a time, `totalLockPoints` only increased by 100.

So, when the rewards are distributed, the denominator is much lower than it is supposed to be — 298 instead of 396. Then, when LP 2 unlocks all their shares, they get $6,652.01 instead of $5,000 - 2/3 of the reward instead of half.

Later, when LP 1 tries to unlock their shares, the logic also tries to send them 2/3 of the reward but fails due to insufficient funds, since only 1/3 of the reward is actually left.

Recommendations

Track the total lockpoints with the same precision as each individual NFT's lockpoints to prevent precision loss. Verify that, after remediation, nothing can violate the invariant that the sum of all lockpoints across all lock NFTs is equal to `totalLockPoints`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 9c5865d3 ↗.

3.11. Wrong reserve ratio returned by `getReserveRatio` when constrained

Target	VaultManager		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The public view function `getReserveRatio` determines, for a given reserve amount, what ratio to reserve between the junior and senior vaults. This is the target reserve ratio if the target reserve ratio results in a split that is reservable. Otherwise, it is the current reserve ratio:

```
function getReserveRatio(uint _reserveAmount)
public view returns (uint256) {
    if (_reserveAmount > 0) {
        uint currentReserveRatio = getCurrentReserveRatio();
        if (
            !_isNormalLiquidityMode(currentReserveRatio) ||
            !junior.hasLiquidity(
                (_reserveAmount * targetReserveRatio) / 100) ||
            !senior.hasLiquidity(
                (_reserveAmount * targetReserveRatio) / 100)
        ) {
            // constrained Liquidity Mode
            return currentReserveRatio;
        }
    }
    return targetReserveRatio;
}
```

However, there are two things wrong with this function. The first is that the call to `senior.hasLiquidity` should use `_reserveAmount - (_reserveAmount * targetReserveRatio / 100)` instead of the same parameter to `junior.hasLiquidity` because we want to check if the senior tranche has the liquidity for the rest of the assets.

The second is that there is no guarantee that the current reserve ratio returned by `getCurrentReserveRatio()` is a split that allows the reservation of that amount of assets. The reserve ratio this returns is just the proportion of assets, whether they are reserved or not:

```
function getCurrentReserveRatio() public view returns (uint256) {
    IERC20 asset = IERC20(junior.asset());
    if (asset.balanceOf(address(senior)) == 0 &&
        asset.balanceOf(address(junior)) == 0) {
        return targetReserveRatio;
    }
    return
        (100 * asset.balanceOf(address(junior))) /
        (asset.balanceOf(address(junior)) +
            asset.balanceOf(address(senior)));
}
```

So, for example, if the junior and senior vaults have the same amount of assets, but all of the junior is reserved and none of the senior is reserved, this will return 50%, which causes the reservation to fail. If it had returned 0% instead, the reservation would succeed, which it should because there are enough assets in the vaults to back the reservation.

Impact

Opening trades can unexpectedly revert in `reserveBalance` even if there should be enough liquidity to reserve the required balance.

A front-runner looking to cancel a trade open transaction by sandwiching it needs much less collateral to do so.

Recommendations

For the first issue, fix the coding mistake to correctly check if the reserved amount requires the constrained liquidity mode.

For the second issue, instead of using the current reserve ratio as the reserve ratio, the quantity that should be used is the current ratio of unreserved funds between the two tranches.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- `b5416aea` ↗
- `9dce46f8` ↗

3.12. Loss-protection tier is reduced for trades that greatly reduce skew

Target	PairInfos		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The loss-protection–tier feature is intended to incentivize traders to submit trades that reduce the current skew, which are trades that inverse the net-demand position of the protocol. However, when a trade is opened, the loss-protection tier is calculated based on the skew after the trade's open interest is applied to the total:

```
function lossProtectionTier(ITradingStorage.Trade memory _trade)
external view override returns (uint) {
    uint openInterestUSDCLong =
        storageT.openInterestUSDC(_trade.pairIndex, 0);
    uint openInterestUSDCShort =
        storageT.openInterestUSDC(_trade.pairIndex, 1);

    uint updatedInterest = _trade.initialPosToken.mul(_trade.leverage);

    if (!_trade.buy) {
        openInterestUSDCShort += updatedInterest;
        uint openInterestUSDCLongPct = (100 * openInterestUSDCLong) /
            (openInterestUSDCLong + openInterestUSDCShort);
        //...snipped loop checks openInterestUSDCLongPct vs longSkewConfig
    } else {
        openInterestUSDCLong += updatedInterest;
        uint openInterestUSDCShortPct = (100 * openInterestUSDCShort) /
            (openInterestUSDCLong + openInterestUSDCShort);
        //...snipped loop checks openInterestUSDCShortPct vs shortSkewConfig
    }
    return 0; // No Protection Tier
}
```

For both long and short opens (buys and sells, respectively), the `updatedInterest` is added to the open interest before calculating the skew.

Impact

This means that a trade that corrects enough skew to move the skew config to a lesser tier will get a lesser degree of loss protection, which is an unexpected result. In the extreme case, if a trader corrects the entire skew with a large transaction, they will get no loss protection. This is counterintuitive to the purpose of the loss-protection feature.

Recommendations

Since loss-protection tiers are a staircase-shaped function, if a trade causes the total open interest to span multiple skew config values, the area under the curve needs to be calculated to determine the total incentive the trader should get. However, currently this incentive is tiered so the trade cannot be awarded half of a tier, so that does not map cleanly.

When we brought up this finding with Avantis, we also noted other concerns with the loss-protection tier feature even when it is working as intended. For example, since loss protection stays with the trade, if a trader creates a trade that is awarded loss protection, and then the protocol skew changes direction, the trader can add any amount of funds to the existing trade using a margin update. This effectively causes the added position to also benefit from the loss protection despite worsening the skew. Also, it is possible to place a temporary trade with a flash loan to intentionally skew the protocol first, before an actual trade is placed, to essentially “buy” loss protection for a fee.

In light of this finding and the other concerns, we recommend rethinking how loss-protection tiers should work to incentivize traders to reduce skew.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `dfc3241f` ↗. This commit adds a per-block open interest limit in order to prevent the use of flash loans to exploit this.

3.13. Tranche trading inflow is much less than outflow in zero skew

Target	TradingCallbacks		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

When a trade is closed at a loss in `_unregisterTrade`, the collateral that is lost is allocated as a reward:

```
if (pnl < 0) {
    storageT.vaultManager().allocateRewards(
        uint(-pnl) + totalFees - _feeAmountToken);
}
```

When a reward is allocated, it is divided between Tranche and VeTranche depending on the amount of locked assets. This seems safe because the expectation is that the reward is not needed in the future — it becomes profit to the LPs.

However, the inflow of funds due to losing trades is actually needed to offset the outflow of funds due to other winning trades. So, this inflow cannot be considered a reward.

Ideally, across the entire protocol, there should be a high volume of trades, and for each asset the total leveraged long position should equal the total leveraged short position across all the open positions (zero skew). If this is the case, then when the price changes, the large inflow of collateral due to the losing trades is roughly equal in magnitude to the large outflow of collateral due to the winning trades.

Therefore, if the inflow is routed both to Tranche and VeTranche, and the outflow that matches the inflow in size is taken only from Tranche, Tranche will quickly run out of money as more and more money gets stuck in the VeTranche rewards.

Impact

In the ideal case where there is high volume and zero skew, when inflow and outflow of USDC is averaged over all the trades in the system, Tranche will quickly run out of money. This causes traders to be unable to open new trades and LPs who do not lock their shares to lose money to LPs that do lock their shares.

Recommendations

Instead of allocating the proceeds obtained from losing positions as rewards, send them only to Tranche with the expectation that other winning positions will need them when they close.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `36126985` ↗.

3.14. Arbitrage opportunities with older price feeds

Target	PriceAggregator		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The protocol uses Pyth oracles to obtain the prices of the tokens. When opening, modifying, or closing a trade, the user must supply a `priceUpdateData` blob, which is obtained from and signed by the off-chain Pyth oracle.

In `PriceAggregator.fulfill`, this data is then passed to the on-chain Pyth contract using `Pyth.updatePriceFeeds` to update the price before the updated price is fetched using `Pyth.getPrice`, and then that price is used for the order.

Since the `priceUpdateData` blob is taken from user input, it could be old price data or the price data for a different price feed. In this case, the update will be rejected by Pyth, which means `getPrice` will return the latest price the on-chain Pyth contract has seen, which is not necessarily the most recent price that exists.

If we assume that this price is not too stale (it is <2 minutes old) but is also not equal to the most recent price, an arbitrage opportunity exists because the user can place orders under the old price while already knowing what the next price is going to be.

Impact

At times of high volatility (high when leverage is considered), this arbitrage opportunity is worth the fees and can be done inside a single transaction, allowing for risk-free arbitrage.

For example, let's say an asset is relatively stable, so its maximum leverage is set to 100x, and then volatility causes its price to increase by 0.1% in one tick. If a user notices that the Pyth oracle has not yet seen the new price, they can take out a flash loan and then call `openTrade` to open a long position at maximum leverage using a stale or incorrect `priceUpdateData`. This causes the old price to be used. Then, they can provide the new `priceUpdateData` and immediately close the trade in the same transaction, letting them book an instant and risk-free profit of 10% of the value of the flash loan.

Recommendations

The issue at hand is challenging to rectify within the constraints of the current contract-trader interface due to the following reasons:

1. Taking `priceUpdateData` from traders is not safe as traders are always able to refuse to give newer data that they nonetheless already have, until a later call in the same block or transaction.
2. If the responsibility for providing `priceUpdateData` is shifted to governance, it would have to call this function at the update frequency of the underlying price feed, likely many times per block for every pair.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- `c3a43939` ↗
- `dc29a740` ↗

Avantis acknowledges the risk of using user-supplied price update data in margin update and limit execution orders, since feasible exploitation of these arbitrage opportunities will require the payment of margin fees on the leveraged position until the price jumps sufficiently, so it is not risk-free.

3.15. The useBackupOnly mode allows undue margin update withdrawals

Target	TradingCallbacks		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function updateMargin allows users to deposit or withdraw USDC from their open positions. When the function is called to withdraw USDC, it first verifies if the new leverage lies in the correct range, and then, if it does, passes control to the fulfill function in PriceAggregator.

If the value of useBackupOnly is set to true in PriceAggregator, the value of price would be zero and it will be pushed to the answers array. This function would then call the callback function updateMarginCallback with the a.price value as zero.

```
function updateMarginCallback(
    AggregatorAnswer memory a
) external override onlyPriceAggregator {
    //...
        int profitP = _currentPercentProfit(_trade.openPrice,
            a.price, _trade.buy, _trade.leverage);
        int pnl = (int(_trade.initialPosToken) * profitP)
            / int(_PRECISION) / 100;
        if (pnl < 0) {
            pnl = (
                pnl * int(
                    aggregator.pairsStorage()
                    .lossProtectionMultiplier(_trade.pairIndex, o.tier)
                )
            ) / 100;
        }
        require((int(_trade.initialPosToken) + pnl) >
            (int(_trade.initialPosToken) *
                int(100 - _WITHDRAW_THRESHOLD_P))
            / 100, "WITHDRAW_THRES_BREACHED");
    }
    storageT.vaultManager().sendUSDCToTrader(_trade.trader, o.amount);
    //...
}
```

```
function _currentPercentProfit(
    uint openPrice,
    uint currentPrice,
    bool buy,
    uint leverage
) private returns (int p) {
    int diff = buy ?
        (int(currentPrice) - int(openPrice)) :
        (int(openPrice) - int(currentPrice));
    int minPnlP = int(_PRECISION) * (-100);
    int maxPnlP = int(_MAX_GAIN_P) * int(_PRECISION);
    p = (diff * 100 * int(_PRECISION.mul(leverage))) / int(openPrice);
    p = p < minPnlP ? minPnlP : p > maxPnlP ? maxPnlP : p;
}
```

If the original trade was a short trade, the returned value of `profitP` would be `100 * int(_PRECISION.mul(leverage))` (max capped to 900%). Therefore, even if the real price is higher (which means the short suffered a loss), the trader can still withdraw their tokens as if the price actually hit zero.

Impact

Users can withdraw much more tokens from their position than they should be allowed to.

Recommendations

The other callback functions delete the pending order and return without making any changes in the trades if the value of `a.price` is zero. The same check here would prevent this issue.

Alternatively, if Finding 3.23. ↗ is resolved by making cancelled orders revert, consider reverting in PriceAggregator instead of calling any callbacks when the price is set to zero because it is not valid.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `a445138c` ↗.

3.16. The applyReferralClose function returns fee with referrer rebate

Target	TradingStorage		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When a trade is being closed, the function `_unregisterTrade` is called to unregister the trade. The function calls `applyReferralClose` to calculate the `referrerRebate` and `feeAfterRebate`. The issue here is that `feeAfterRebate` includes the referrer rebate too.

The function `applyReferralClose` is responsible to calculate `feeAfterRebate` and `referrerRebate`.

```
function applyReferralClose(
    address _trader,
    uint _fees,
    uint _leveragedPosition
) public override onlyTrading returns (uint, uint) {
    (uint traderFeePostDiscount, address referrer, uint referrerRebate) =
        referral.traderReferralDiscount(_trader, _fees);

    if (referrer != address(0)) {
        rebates[referrer] += referrerRebate;
        emit TradeReferred(
            _trader,
            referrer,
            _leveragedPosition,
            traderFeePostDiscount,
            _fees - traderFeePostDiscount,
            referrerRebate
        );
        return (traderFeePostDiscount, referrerRebate);
    }
    return (_fees, referrerRebate);
}
```

Here the value `traderFeePostDiscount` includes the `referrerRebate`, which should be subtracted from it before it is returned.

Impact

The `feeAfterRebate` is used to allocate rewards using the vault manager. If the `referrerRebate` is not subtracted from it, more rewards would be allocated as compared to what is available.

Recommendations

Subtract `referrerRebate` from `traderFeePostDiscount` before returning the value in `applyReferralClose`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `bd5cb1f1` ↗.

3.17. Bot latency may prevent execution of limit-close orders

Target	TradingCallbacks		
Category	Code Maturity	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

When a limit-close order is executed via `executeLimitOrder`, the callback function checks if the current price (`a.price`) is in the correct range depending on the type of the limit order.

```
function executeLimitCloseOrderCallback(AggregatorAnswer memory a)
external override onlyPriceAggregator {
    //...
    if (o.orderType == ITradingStorage.LimitOrder.LIQ) {
        uint liqPrice = pairInfos.getTradeLiquidationPrice(
            t.trader,
            t.pairIndex,
            t.index,
            t.openPrice,
            t.buy,
            v.posUSDC,
            t.leverage
        );
        v.reward = (t.buy ? a.price <= liqPrice : a.price >= liqPrice) ?
            (v.posToken * liqFeeP) / 100 : 0;
    } else {
        v.reward = (o.orderType == ITradingStorage.LimitOrder.TP &&
            t.tp > 0 &&
            (t.buy ? a.price >= t.tp : a.price <= t.tp)) ||
            (o.orderType == ITradingStorage.LimitOrder.SL &&
                t.sl > 0 &&
                (t.buy ? a.price <= t.sl : a.price >= t.sl))
            ? (
                v.posToken.mul(t.leverage) *
                aggregator.pairsStorage().pairLimitOrderFeeP(t.pairIndex)
            ) / 100 / _PRECISION
            : 0;
    }
    //...
}
```

It might be possible that the bot executes the limit-close order with a minor delay, during which the price falls out of the correct range and the order is not executed.

For example, for a stop loss to be triggered, the stop loss should be greater than the current price if the order is a buy order. The bot verifies the price at every block, and if it finds an order where stop loss is greater than the current price, it executes a limit close on that trade. But due to some latency in the bot, it might be possible that it calls the `executeLimitOrder` at the time the price goes above the stop loss — due to which, it will not be triggered.

Impact

Limit-close orders might not be executed in case of bot latency. In the worst case scenario, it means that stop loss, take profits, and liquidation will not be executed.

Recommendations

We recommend tracking the highs and lows between the time an order is placed (or the SL/TP is updated) until the time the `executeLimitOrder` is called, and only execute the order if the `liqPrice` / `t.tp` / `t.sl` (depending on the type of order) falls in between that range.

Remediation

This issue has been acknowledged by Avantis Labs, Inc.. Avantis plans to remediate this issue through more robust backend infrastructure for liquidation bots.

3.18. Referrer-code transfers overwrite recipient codes and misalign tiers

Target	Referral		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The function `setCodeOwner` allows an account that owns a referral code to transfer that code to another account:

```
function setCodeOwner(bytes32 _code, address _newAccount) external {
    require(_code != bytes32(0), "ReferralStorage: invalid _code");

    address account = codeOwners[_code];
    require(msg.sender == account, "ReferralStorage: forbidden");

    codeOwners[_code] = _newAccount;

    delete codes[account];
    codes[_newAccount] = _code;

    emit SetCodeOwner(msg.sender, _newAccount, _code);
}
```

However, `_newAccount` may already have a code or may not want to receive the code.

Additionally, the transfer process does not update the referrer tier of the sender or recipient, so if the recipient did not have a code, they will stay at tier zero due to the uninitialized field rather than being set to tier one. And either way, the sender will keep their tier, despite having given away the code, until they register another code.

Impact

Anyone can overwrite anyone else's referrer code in the `codes` mapping to their own referrer code.

Also, tiers and referral codes will become out of sync upon a transfer.

Recommendations

Rework the code-transfer process to include a step where the recipient affirms the transfer before it actually takes place. Also, have the code-transfer process transfer the tier or associate tiers with referrer accounts instead of codes.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- 0524a688 ↗
- 73c26d9a ↗

3.19. Delayed force unlock causes reward insolvency

Target	VeTranche		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

Keeper bots are responsible for calling `forceUnlock` in order to unlock expired lock positions after their lock time has passed. However, if `forceUnlock` is not called on an expired lock position, it continues earning rewards.

Impact

If the keeper bot fails to `forceUnlock` an expired position and a rewards distribution happens, the expired position will accrue undue rewards.

Recommendations

To fix this on chain while keeping constant-time rewards allocations and distributions, the cumulative rewards-per-sharepoint value can be saved in an array instead of an updated state variable, alongside the time of the rewards distribution. Then, unlock can check if a reward distribution has happened since the lock expired — if it did, then either binary search for the value at expiration time or require that the user supply it as a parameter.

Remediation

This issue has been acknowledged by Avantis Labs, Inc.. Avantis remediated this issue by ensuring that the unlock bots have reasonably low latency.

3.20. Price impact is not tracked cumulatively

Target	PairInfos		
Category	Protocol Risks	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The price impact of a trade is calculated in `getTradePriceImpact` which calls `getTradePriceImpactPure`:

```
function getTradePriceImpact(
    uint openPrice,
    uint pairIndex,
    bool long,
    uint tradeOpenInterest
) external view override
returns (uint priceImpactP, uint priceAfterImpact) {
    (priceImpactP, priceAfterImpact) = getTradePriceImpactPure(
        openPrice,
        long,
        tradeOpenInterest,
        long ? pairParams[pairIndex].onePercentDepthAbove :
            pairParams[pairIndex].onePercentDepthBelow
    );
}

function getTradePriceImpactPure(
    uint openPrice,
    bool long,
    uint tradeOpenInterest,
    uint onePercentDepth
) public pure returns (uint priceImpactP, uint priceAfterImpact) {
    if (onePercentDepth == 0) {
        return (0, openPrice);
    }

    priceImpactP = (tradeOpenInterest * _PRECISION) / onePercentDepth;
    uint priceImpact = (priceImpactP * openPrice) / _PRECISION / 100;
    priceAfterImpact = long ? openPrice + priceImpact :
        openPrice - priceImpact;
}
```

The price impact is calculated based only on the Pyth price `openPrice` and the size of the position being opened `tradeOpenInterest`. Using these parameters, the price impact is approximated linearly using a governance-set one percent depth above or below parameter.

Impact

Since the price impact is not cumulative across multiple successive trades, instead of placing one large trade with large price impact, a trader should instead split it up into multiple smaller trades. Allowing a sophisticated trader to experience a much lower price impact by splitting up their trade adds considerable risk because the price impact is then cumulatively not correctly modeled.

Recommendations

Cumulatively track the price impact, resetting it after a new price is obtained from the oracle so that there is little or no benefit to splitting up a trade into smaller trades.

Remediation

This issue has been acknowledged by Avantis Labs, Inc.. Avantis is changing the price impact formulation as per their ongoing economic modelling.

3.21. Loss protection reduces the -100% cap on losses

Target	TradingCallbacks		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

When a trade is closed, `_currentPercentProfit` is called to get the percent profit:

```
function _currentPercentProfit(
    uint openPrice,
    uint currentPrice,
    bool buy,
    uint leverage
) private pure returns (int p) {
    int diff = buy ? (int(currentPrice) - int(openPrice))
                   : (int(openPrice) - int(currentPrice));
    int minPnlP = int(_PRECISION) * (-100);
    int maxPnlP = int(_MAX_GAIN_P) * int(_PRECISION);
    p = (diff * 100 * int(_PRECISION.mul(leverage))) / int(openPrice);
    p = p < minPnlP ? minPnlP : p > maxPnlP ? maxPnlP : p;
}
```

Note that the minimum this can return is -100%. Then, the return value is passed as `_percentProfit` to `_unregisterTrade`, which calls `PairInfos.getTradeValue`, which fetches the actual loss-protection percentage and then calls `getTradeValuePure`:

```
function getTradeValuePure(
    uint collateral,
    int percentProfit,
    uint rolloverFee,
    uint closingFee,
    uint lossProtection
) public pure returns (uint, int, uint) {
    int pnl = (int(collateral) * percentProfit) / int(_PRECISION) / 100;
    if (pnl < 0) {
        pnl = (pnl * int(lossProtection)) / 100;
    }
    int fees = int(rolloverFee) + int(closingFee);
    int value = int(collateral) + pnl - fees;
    if (value <= (int(collateral) * int(100 - _LIQ_THRESHOLD_P)) / 100) {
        value = 0;
    }
    return (value > 0 ? uint(value) : 0, pnl, uint(fees));
}
```

So, the loss protection is applied after the -100% minimum.

Impact

A trade with a loss-protection tier has a cap on the amount of loss that is greater than complete loss. This means that the trader will always get back an amount, no matter how much their position loses.

Recommendations

Instead of constraining the loss to -100% in `_currentPercentProfit`, consider handling the case of unlimited losses in a higher-level function so that it is aware of the loss-protection tier.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- `5399494a` ↗
- `21e210ea` ↗

3.22. Variable reuse causes `totalPrincipalDeposited` miscalculation

Target	Tranche		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The function `_updateNegativePrincipal` is an internal function used to update principal deposits during withdrawals or transfers. The value of `principalAssetDiff` is first calculated using the amount of shares that are to be removed. Then `principalAssetsDeposited[user]` and `totalPrincipalDeposited` is decreased, as shown below:

```
function _updateNegativePrincipal(address user, uint256 shares) internal {
    uint256 principalAssetDiff =
        (shares * principalAssetsDeposited[user])
        / principalSharesDeposited[user];
    principalAssetsDeposited[user] -=
        principalAssetDiff < principalAssetsDeposited[user] ?
        principalAssetDiff : principalAssetsDeposited[user];
    totalPrincipalDeposited -=
        principalAssetDiff < principalAssetsDeposited[user] ?
        principalAssetDiff : principalAssetsDeposited[user];
    principalSharesDeposited[user] -= shares;
}
```

Here, the value of `principalAssetsDeposited[user]` and `totalPrincipalDeposited` are supposed to be decreased by the same amount. As the value of `principalAssetsDeposited[user]` is updated first, this updated value will be used to decrease the value of `totalPrincipalDeposited`, as opposed to the old value. This would lead to `principalAssetsDeposited[user]` and `totalPrincipalDeposited` decreased by different amounts.

Impact

The variable `totalPrincipalDeposited` is used to keep track of total deposited principal and total earnings, which would be incorrect in this case.

Recommendations

The value to be decreased could be stored in a different local variable, and then this value could be used decrease from both the variables.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `8cf936e9` ↗. The Tranche asset tracking feature was removed and moved to off-chain logic.

3.23. Governance fee charged without market-order placement

Target	TradingCallbacks		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

A market order can be cancelled in the callback for various reasons, including if `useBackupOnly` is true, so the value of `a.price` is zero, the execution price is outside slippage parameters, or the trade is not within exposure limits. In this case, the USDC is transferred back to the user after the dev and governance fees are deducted.

Charging the user governance fees in case the backup oracle is used does not seem fair to the users of the protocol. Additionally, a front-runner can sandwich a market open on the mempool with a large trade that consumes all of the open interest, causing the market order to be cancelled. If the back-run side of the sandwich closes the same trade, the price does not change, so there is no risk to the front-runner — but the fees add to the LP returns.

Impact

Users are charged an unfair fee amount.

Recommendations

We recommend removing this fee when a market order is cancelled and instead return to the trader all the USDC or revert.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 36126985 ↗.

3.24. One account can register multiple referral codes

Target	Referral		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The external function `registerCode` allows any account to register a referral code:

```
function registerCode(bytes32 _code) external {
    require(_code != bytes32(0), "ReferralStorage: invalid _code");
    require(codeOwners[_code] == address(0),
        "ReferralStorage: code already exists");

    codeOwners[_code] = msg.sender;
    codes[msg.sender] =  _code;
    referrerTiers[msg.sender] = _DEFAULT_TIER_ID;

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

One account should only have one referral code, but this function does not check that the account does not already have a code registered.

Impact

One account can register multiple referral codes.

Recommendations

Add a check to revert if the account already has a referral code.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 13bb96c4 ↗.

3.25. Vault manager withdrawals cannot access the entire junior tranche

Target	VaultManager		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

In VaultManager, the internal function `_sendUSDCToTrader` sends USDC from the vault to the trader, or, in extreme cases (as noted by the comment), when the vault runs out from the tranches to the trader. It is called when the trader is due USDC due to modifying or closing a position.

```
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 juniorUSDC = (getLossMultiplier() * difference *
            getReserveRatio(0)) / 100 / 100;
        juniorUSDC = (juniorUSDC > difference) ? difference : juniorUSDC;

        uint256 seniorUSDC = difference - juniorUSDC;

        junior.withdrawAsVaultManager(juniorUSDC);
        senior.withdrawAsVaultManager(seniorUSDC);
    }

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

When losses need to be taken from the vaults, it calculates `juniorUSDC` as a proportion of the difference needed. Note that the call to `getLossMultiplier()` and `getReserveRatio(0)` both return constant percentages, the base multiplier, and the target reserve ratio respectively, so the ternary that follows is always false. Then, `seniorUSDC` is the rest of the difference.

However, this logic means that a percentage of the junior tranche is never withdrawn, whereas all of the senior tranche can be withdrawn.

Impact

Large trade-closing transactions can revert due to insufficient funds in the senior tranche, because funds reserved in the junior tranche cannot be accessed due to this effect. Additionally, for larger awards, this makes the junior tranche safer than the senior tranche, violating economic assumptions.

Recommendations

Correctly calculate the proportion of funds taken from junior and senior tranches such that the junior tranche is always more risky than the senior tranche and the current balances of the tranches are considered.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- `874dc022` ↗
- `f9342644` ↗

3.26. The maxRedeem function should comply with ERC-4626

Target	Tranche		
Category	Protocol Risks	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

Tranche should be an ERC-4626–compliant contract. In the ERC-4626 specification, the `maxRedeem` function should return the maximum amount of shares that can be redeemed, keeping in mind all redemption limits.

One of the redemption limits that apply to withdrawals is the utilization ratio reserve requirement. Since `maxRedeem` is currently not overridden from the underlying ERC-4626 contract, it doesn't consider this requirement, and it needs to be aware of this limit to adjust the return value down if applicable.

Impact

Other contracts or external front-ends expecting ERC-4626 compliance can unexpectedly revert.

Recommendations

Override `maxRedeem` with an implementation that considers the reserve ratio requirement.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- `1fcf6cd8` ↗
- `f9342644` ↗

3.27. Incorrect access control of `setVaultManager` causes update lockout

Target	VeTranche		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `setVaultManager` function is used in VeTranche to update the address of the vault manager contract:

```
function setVaultManager(address _vaultManager) external onlyManager {
    require(_vaultManager != address(0), "ADDRESS_INVALID");
    vaultManager = IVaultManager(_vaultManager);
}
```

This function is `onlyManager`, so it can only be called by the vault manager. However, the current version of the vault manager contract never calls this function.

Impact

The vault manager cannot be updated.

Recommendations

Change this to `onlyGov`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `af954df9` ↗.

3.28. Trader contract can bypass max trades per pair

Target	Trading		
Category	Protocol Risks	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

There is a limit on the number of trades a trader can have open:

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

However, this limit can be bypassed by operating from multiple trading accounts or by using a contract that splits requested trades across multiple deployed proxies.

Impact

This limit can be bypassed for sophisticated traders.

Recommendations

We recommend removing this limit to equalize the playing field between traders using the front-end and sophisticated traders who deploy contracts to instantiate trades.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `ab1962d7` ↗.

3.29. Limit-order timelock is not initialized on open

Target	TradingCallbacks		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

In Trading, the `limitOrdersTimelock` parameter is intended to prevent changing the take-profit or stop-loss parameters of a trade too often after they were last changed. This is tracked by the `TradeInfo.tpLastUpdated` and `TradeInfo.slLastUpdated` struct fields, respectively.

However, when this struct is initialized during the creation of a trade in `_registerTrade`, these fields, the second and third ones in the constructor, are set to zero:

```
ITradingStorage.TradeInfo(
    _trade.initialPosToken.mul(_trade.leverage),
    0,
    0,
    false,
    pairInfos.lossProtectionTier(_trade)
)
```

Impact

After a limit order is created, the first SL and TP updates can happen at any time, even before the timelock period has elapsed.

Recommendations

Instead of initializing them to zero, initialize them to `block.number`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `5dca8a6d` ↗.

3.30. Partial closes emit incorrect value in `TradeReferred` event

Target	TradingCallbacks		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When a trade is partially closed, `_unregisterTrade` calls `storageT.applyReferralClose` to apply the referral close:

```
function _unregisterTrade(
    ITradingStorage.Trade memory _trade,
    int _percentProfit,
    uint _collateral,
    uint _feeAmountToken, // executor reward
    uint _lpFeeToken,
    uint _tier
) private returns (uint usdcSentToTrader) {
    //...
    (uint feeAfterRebate, uint referrerRebate) =
        storageT.applyReferralClose(
            _trade.trader,
            _lpFeeToken,
            _trade.initialPosToken.mul(_trade.leverage)
        );
```

However, the third parameter to `applyReferralClose` should be the size of the leveraged position that was closed, instead of the total leveraged position.

In `applyReferralClose`, the third parameter is only used in the emitted `TradeReferred` event.

Impact

The emitted `TradeReferred` event for a partial close will have an incorrect closed leveraged position size.

Recommendations

Make this `_collateral.mul(_trade.leverage)`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `f4ef1c84` ↗.

3.31. Function openTrade lacks incorrect-payment sanity checks

Target	Trading		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `openTrade` is called in order to open both market and limit orders. Opening a market order requires paying the Pyth oracle fee in ETH and a valid `priceUpdateData`. Opening a limit order means the caller needs to pay an additional `_executionFee`.

However, for both market and limit orders, the value sent with the transaction is not checked. If a user accidentally sends more value than is needed for the Pyth update or sends any value with a limit transaction, it will be stuck in the contract.

Also, for market orders, the `_executionFee` parameter is still added to the amount of USDC transferred from the user, but then the variable is not used anywhere else, so the funds remain stuck in the Trading contract.

Impact

User error while calling the `openTrade` function can cause USDC or ETH to be locked.

Recommendations

Ensure that the value sent with the transaction is correct, whether or not the Pyth oracle is consulted. Ensure that the `_executionFee` is not taken from the user if they open a market order, or revert if it is nonzero on a market order.

Remediation

The remediation for Finding 3.14. ↗ removed immediately-executing open orders.

3.32. Timestamp updated in memory instead of storage

Target	TradingCallbacks		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The callback function (`updateSlCallback`) used to update the stop loss also updates the `Trade.timestamp` value, but this value is currently updated in memory and not in storage.

```
function updateSlCallback(AggregatorAnswer memory a)
external override onlyPriceAggregator {
    //...

    ITradingStorage.Trade memory t =
        storageT.openTrades(o.trader, o.pairIndex, o.index);
    if (
        // ...
    ) {
        storageT.updateSl(o.trader, o.pairIndex, o.index, o.newSl);
        t.timestamp = block.timestamp;
        emit SlUpdated(a.orderId, o.trader, o.pairIndex,
            o.index, o.newSl, block.timestamp);
    }

    aggregator.unregisterPendingSlOrder(a.orderId);
}
```

The line `t.timestamp = block.timestamp;` only writes to memory, and `t` is not used after that. This results in no such change of timestamp being stored for that trade.

Impact

The timestamp will not be updated in storage.

Recommendations

We recommend using storage instead of memory so that the timestamp is updated.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 20b78585 ↗.

3.33. Withdraw to different receiver imbalances stats

Target	Tranche		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

When withdrawing from a Tranche, the owner may specify a different receiver for the withdrawn assets per the ERC-4626 specification. When this happens, the `_withdraw` internal function tracks the statistics as follows:

```
function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal virtual override {
    //...

    // use original asset / share ratio and subject the relative asset amount
    if (receiver != owner) {
        _updateNegativePrincipal(owner, shares);

        // gifts are treated as deposits
        principalAssetsDeposited[receiver] += (assets - fee) * _PRECISION;
        totalPrincipalDeposited += (assets - fee) * _PRECISION;
        principalSharesDeposited[receiver] += shares;
    } else if (principalSharesDeposited[receiver] > 0) {
        _updateNegativePrincipal(receiver, shares);
    }
}
```

However, gifts should not be treated as deposits, since the assets are being withdrawn.

Impact

The `principalAssetsDeposited` and `totalPrincipalDeposited` statistics are incorrectly changed after a withdraw to a receiver different from the owner.

Recommendations

Fix this logic to correctly calculate the statistics.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `8cf936e9` ↗. The Tranche statistics were removed.

3.34. Tranche name hardcodes junior symbol

Target	Tranche		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

In the initializer for Tranche, the ERC-20 initializer is called with `j` concatenated onto the name of the underlying asset:

```
__ERC20_init_unchained(
    string(abi.encodePacked(trancheName,
        abi.encodePacked(" Tranche ", ERC20(__asset).name()))),
    string(abi.encodePacked("j", ERC20(__asset).symbol()))
);
```

However, this means that both the junior and senior tranches use `j` and then the underlying asset symbol.

Impact

Holders of senior tranche tokens may be misled by the incorrect name.

Recommendations

Set the symbol depending on the name of the tranche.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `9e9cfb73` ↗.

3.35. Function distributeRewards does not need totalLockPoints

Target	VeTranche		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The distributeRewards function distributes rewards sent to the VeTranche, and it takes in a _totalLockPoints parameter that is expected to equal the contract's totalLockPoints state parameter. If it ever is not equal to that, then a different amount of rewards will be distributed after all shareholders claim rewards, causing either locked funds or insolvency depending on the direction.

Impact

There is no impact, because the VaultManager always correctly calls getTotalLockPoints() to get this quantity to pass back into VeTranche.

However, this is a footgun, since if VeTranche is passed a _totalLockPoints that differs from its state totalLockPoints, it immediately becomes insolvent.

Additionally, it would save gas to have VeTranche read this parameter from its own state, instead of having the information pass from VeTranche to VaultManager back to VeTranche.

Recommendations

Remove this parameter and use the state variable totalLockPoints in VeTranche wherever the value is required.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 18b386c5 ↗.

3.36. Incorrect ternary operator precedence in limit-open-order callback

Target	TradingCallbacks		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In `executeLimitOpenOrderCallback`, there is a conditional that determines if the trade succeeds:

```
if (
    t == IExecute.OpenLimitOrderType.MARKET
        ? (a.price >= o.minPrice && a.price <= o.maxPrice)
        : (
            t == IExecute.OpenLimitOrderType.REVERSAL
                ? (o.buy ? a.price >= o.maxPrice : a.price <= o.minPrice)
                : (o.buy ? a.price <= o.maxPrice : a.price >= o.minPrice)
        ) && _withinExposureLimits(o.trader, o.pairIndex,
            o.positionSize.mul(o.leverage))
) {
    ITradingStorage.Trade memory finalTrade = _registerTrade(
    //...
```

The `_withinExposureLimits` check should happen for all order types; however, if `t` is `MARKET`, then it is not executed because the ternary operator `?:` has lower precedence than the `&&`.

Impact

There is no impact currently because the first branch of the ternary is never executed — this is the limit order callback, so `t` is never a market order.

Recommendations

Since `t` is not ever a market order in this callback, we recommend removing the ternary to prevent this code from being reused in an exploitable way.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- `a6883fac` ↗
- `21e210ea` ↗

3.37. Unused vault-fee parameter must be zero

Target	VaultManager		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `receiveUSDCFromTrader` function has a parameter for vault fees in the implementation of the function:

```
function _receiveUSDCFromTrader(address _trader, uint _amount,
uint _vaultFee) internal {
    storageT.transferUSDC(address(storageT), address(this), _amount);

    if (_vaultFee > 0) totalRewards += _vaultFee;
    emit USDCReceivedFromTrader(_trader, _amount, _vaultFee);
}
```

The vault fee is not taken from the user; it is only allocated as a reward. This means that the function expects the caller to be responsible for transferring the vault fee, which is confusing because this function is responsible for transferring the collateral itself.

Impact

There is no impact because this function is only ever called with a zero vault fee.

Recommendations

We recommend either removing this parameter altogether or adding it to the amount of USDC transferred from storage so that future changes are less likely to create issues.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `f8af8cb2` ↗.

3.38. Execute trigger check never fails due to atomicity

Target	Execute		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Execute contract stores the block number and the first caller to trigger a limit order. However, because transactions are atomic, there is no chance for a second trigger to start between the first trigger and its callback.

Impact

In `Trading.executeLimitOrder`, the call to `executor.triggered` will always return false, and the registration and unregistration of the first bot to trigger the limit in Execute's storage is wholly unnecessary.

Recommendations

To save gas, the entire Execute contract can be replaced with transferring the execution fee to the sender in `executeLimitOrder` as well as ensuring that the callback either successfully executes the trigger or reverts.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `aadd8210` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Referral code incentives are misaligned

The referral code system allows for a fee discount to be given to traders. However, since all the referral codes are on chain, if there is any referral code with the best fee discount, profit-maximizing traders will use that referral code regardless of who actually referred them.

Additionally, the referral code system allows for a rebate to be given to the referrer. However, this rebate can be gamed by the referral code owner. If the owner sets up a contract that reimburses back to the trader the referrer rebates that it gets, minus perhaps a profit margin, they can ask profit-maximizing traders to use their referral code for the maximum effective discount.

4.2. VeTranche NFT info should be struct

When a VeTranche NFT is minted and shares are locked, parameters are set across a few maps, all keyed by the token ID of the new token:

```
tokensByTokenId[nextTokenId] = shares;
lockTimeByTokenId[nextTokenId] = endTime;
lockStartTimeByTokenId[nextTokenId] = block.timestamp;
rewardsByTokenId[nextTokenId] = 0;
lockMultiplierByTokenId[nextTokenId] =
    getLockPoints(endTime - block.timestamp);
```

This code would be more readable and save more gas (due to saving `keccak` calls) if these were instead fields of a struct and if a single mapping that goes from token ID to the struct stored all of this information.

4.3. No way to remove user from Trading whitelist

The Trading contract has a feature to whitelist traders. There is no way to remove a user from this whitelist though, so if a user needs to be removed, the contract must be upgraded to facilitate that.

4.4. Pyth price can become negative or erratic

Avantis treats the price as an unsigned quantity, but according to the Pyth documentation, the price is actually signed:

```
struct Price {
    // Price
    int64 price;
    // Confidence interval around the price
    uint64 conf;
    // Price exponent
    int32 expo;
    // Unix timestamp describing when the price was published
    uint publishTime;
}
```

The price of real-world equities is sometimes negative (for example, this happened to the price of oil in 2021). In the event this actually happens, when cast to an unsigned quantity, Avantis will instead assume it is very large, causing longs to win when shorts should have won.

Additionally, when a stock split happens, such as the TSLA stock split in 2022, the Pyth price feed changes to reflect the new price after off-chain notifications and announcements. If this happens and the Avantis governance does not notice and step in, many traders will suffer incorrect gains/losses due to the sudden predictable discontinuity in price.

4.5. Tranche has unnecessary `ReentrancyGuard`

The Tranche contract inherits from ReentrancyGuardUpgradeable, but none of its functions are marked with the `nonReentrant` modifier, so the base contract is not doing anything.

4.6. Checks-effects-interactions pattern broken

The function `claimRebate` transfers the USDC to the user before setting `rebates` to zero.

```
function claimRebate() external {
    usdc.transfer(msg.sender, rebates[msg.sender]);
    rebates[msg.sender] = 0;
}
```

Although this is currently not a security issue, if the protocol decides to use any other token in the future with hooks on transfer, it would be a security risk.

4.7. Typos

We noticed several minor typos that do not affect code functionality, but nevertheless should be fixed.

- In TradingStorage, the parameter to `setReferral` is `_refferal`.
- In PairStorage, the gov-only function name is `udpateSkewOpenFees`.
- In ITradingStorage, `updateType` is an enum, so it should be capitalized.
- In VeTranche, in the `unlock` and `forceUnlock` functions, there is a duplicate `delete lockStartTimeByTokenId[tokenId];`.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: Execute.sol

Function: `claimTokens()`

This claims the pending token reward for the caller.

Branches and code coverage

Intended branches

- If the caller has some tokens to claim, transfer these tokens to the caller.
  ☑ Test coverage

Negative behavior

- Revert if `tokensToClaim` for `msg.sender` is zero.
  □ Negative test

Function call analysis

- `ICallbacks(this.storageT.callbacks()).transferFromVault(msg.sender, tokens)`
  - **What is controllable?** `msg.sender` and `tokens`.
  - **If the return value is controllable, how is it used and how can it go wrong?** No return values.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

5.2. Module: Referral.sol

Function: `registerCode(byte[32] _code)`

This registers a referral code for the sender — to be called by referrer.

Inputs

- `_code`
  - **Control**: Fully controlled by caller.
  - **Constraints**: The `_code` should not be `bytes32(0)` and `codeOwners` should not be already registered.
  - **Impact**: The referral code to register.

Branches and code coverage

Intended branches

- The `codeOwners`, `codes`, and `referrerTiers` is correctly updated.
  ☑ Test coverage

Negative behavior

- Revert if `_code` is `bytes32(0)`.
  □ Negative test
- Revert if `codeOwners[_code] != address(0)`.
  □ Negative test

Function call analysis

No external function calls found.

Function: `setCodeOwner(byte[32] _code, address _newAccount)`

This changes the owner of a referral code — to be called by the code owner.

Inputs

- `_code`
  - **Control**: Fully controlled by caller.
  - **Constraints**: Must not be an empty code `(bytes32(0))`.
  - **Impact**: The referral code to change ownership.
- `_newAccount`
  - **Control**: Fully controlled by caller.
  - **Constraints**: None.
  - **Impact**: The new owner address.

Branches and code coverage

Intended branches

- Updates the `codeOwner` of the provided `_code`.
  - ☑ Test coverage

Negative behavior

- Revert if `_code` is an empty code `(bytes32(0))`. (Prevents changing ownership for an invalid code.)
  - □ Negative test
- Revert if the caller is not the current owner of the code. This ensures that only the current owner can change ownership.
  - □ Negative test

Function call analysis

No external function calls found.

Function: `setTraderReferralCodeByUser(byte[32] _code)`

This sets the trader's referral code — callable by user.

Inputs

- `_code`
  - **Control**: Fully controlled by caller.
  - **Constraints**: None.
  - **Impact**: The referral code.

Branches and code coverage

Intended branches

- The referral code is correctly set for `msg.sender`.
  - ☑ Test coverage

Negative behavior

N/A.

Function call analysis

No external function calls found.

5.3. Module: TradingStorage.sol

Function: `claimRebate()`

This allows a referrer to claim their rebate.

Branches and code coverage

Intended branches

- If there is some value in rebates for `msg.sender`, transfer it to the caller and set this value to zero.
  - ☑ Test coverage

Negative behavior

N/A.

Function call analysis

- `this.usdc.transfer(msg.sender, this.rebates[msg.sender])`
  - **What is controllable?** `msg.sender` and `this.rebates[msg.sender]`.
  - **If the return value is controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

5.4. Module: Trading.sol

Function: `cancelOpenLimitOrder(uint256 _pairIndex, uint256 _index)`

This cancels an open limit order.

Inputs

- `_pairIndex`
  - **Control**: Fully controlled by caller.
  - **Constraints**: None.
  - **Impact**: The index of the trading pair.
- `_index`
  - **Control**: Fully controlled by caller.
  - **Constraints**: None.
  - **Impact**: The index of the order.

Branches and code coverage

Intended branches

- Unregisters the open limit order.
  - ☑ Test coverage
- Transfers the USDC to the trader.
  - ☑ Test coverage

Negative behavior

- Revert if `block.number - o.block` is less than `limitOrdersTimelock`.
  - □ Negative test

Function call analysis

- `this.storageT.getOpenLimitOrder(msg.sender, _pairIndex, _index)`
  - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the open limit order; this limit order is updated and later stored in storage.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.storageT.transferUSDC(address(this.storageT), msg.sender, o.positionSize + o.executionFee)`
  - **What is controllable?** `msg.sender`, `o.positionSize`, and `o.executionFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Transfers the USDC balance associated with the canceled order back to the caller — no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.unregisterOpenLimitOrder(msg.sender, _pairIndex, _index)`
  - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `closeTradeMarket(uint256 _pairIndex, uint256 _index, uint256 _amount, bytes[] priceUpdateData)`

This closes a trade using market execution.

Inputs

- `_pairIndex`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index of the trading pair for the open trade.
- `_index`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index of the open trade.
- `_amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The collateral by which to update the margin.
- `priceUpdateData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Pyth price update data.

Branches and code coverage

Intended branches

- Generates an new `orderId` for the close trade market and calls `fulfill` in the price aggregator.
    - ☑ Test coverage
- The `fulfill` in the price aggregator fetches the price from the oracle and then calls the callback function in `TradingCallbacks`.
    - ☑ Test coverage
- The callback function unregisters the trade and unregisters the pending market order.
    - ☑ Test coverage

Negative behavior

- Revert if pending orders are more than or equal to the max pending market order value.
    - □ Negative test
- Revert if the market order is already closed.
    - □ Negative test
- Revert if the leverage of the trade is zero.
    - □ Negative test

Function call analysis

- `this.storageT.openTrades(msg.sender, _pairIndex, _index)`
    - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade; incorrect values may lead to incorrect trade information retrieval.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.openTradesInfo(msg.sender, _pairIndex, _index)`
    - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves additional information about the open trade; incorrect values may lead to incorrect information retrieval.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert; no reentrancy scenarios.
- `this.storageT.pendingOrderIdsCount(msg.sender)`
    - **What is controllable?** `msg.sender`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the count of pending orders for the user.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.maxPendingMarketOrders()`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the max pending market orders.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.priceAggregator().getPrice(_pairIndex, OrderType.MARKET_CLOSE)`
    - **What is controllable?** `pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `orderId` of the current order.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.storePendingMarketOrder(PendingMarketOrder(Trade(msg.sender, t.pairIndex, 0, 0, t.positionSizeUSDC, 0, t.buy, t.leverage, t.tp, t.sl, 0), 0, t.openPrice, _slippageP), orderId, True)`
    - **What is controllable?** `msg.sender`, `t.pairIndex`, `t.positionSizeUSDC`, `t.buy`, `t.leverage`, `t.tp`, `t.sl`, `t.openPrice`, and `_slippageP`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Stores the pending market order — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.storageT.priceAggregator().fulfill{value: msg.value}`
    - **What is controllable?** `msg.value`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Fulfills the update margin order — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `executeLimitOrder(ITradingStorage.LimitOrder _orderType, address _trader, uint256 _pairIndex, uint256 _index, bytes[] priceUpdateData)`

This executes a limit order (either open or close).

Inputs

- `_orderType`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The type of limit order (OPEN, CLOSE, TP, SL, or LIQ).
- `_trader`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The address of the trader.
- `_pairIndex`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index of the trading pair.
- `_index`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index of the order.
- `priceUpdateData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Pyth price update data.

Branches and code coverage

Intended branches

- If order type is OPEN, the fulfill function calls `executeLimitOpenOrderCallback`; otherwise, it calls `executeLimitCloseOrderCallback`.

  - ☑ Test coverage
- For crypto trading pairs, use `onePercentP` to calculate trade price impact.
  - ☑ Test coverage
- Registers the trade and unregisters the open limit order.
  - ☑ Test coverage

Negative behavior

- Revert if the limit-order type is OPEN and no corresponding open limit order is found.
  - □ Negative test
- Revert if the limit-order type is CLOSE, SL, or LIQ and no corresponding trade is found.
  - □ Negative test
- Revert if the SL order type is specified without a valid stop-loss price.
  - □ Negative test
- Revert if the LIQ order type is specified with a stop loss that would be triggered.
  - □ Negative test

Function call analysis

- `this.storageT.hasOpenLimitOrder(_trader, _pairIndex, _index)`
  - **What is controllable?** `_trader`, `_pairIndex`, and `_index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks for the existence of an open limit order — returns true if trade exists.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.openTrades(_trader, _pairIndex, _index)`
  - **What is controllable?** `_trader`, `_pairIndex`, and `_index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the open trade — incorrect values may lead to incorrect trade retrieval.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._getTradeLiquidationPrice(t) -> this.pairInfos.getTradeLiquidationPrice( t.pairIndex, t.index, t.openPrice, t.buy, t.initialPosToken, t.leverage)`
  - **What is controllable?** `t.trader`, `t.pairIndex`, `t.index`, `t.openPrice`, `t.buy`, `t.initialPosToken`, and `t.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Calculates the trade-liquidation price; incorrect values may lead to incorrect liquidation price calculation.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.storageT.priceAggregator()`
  - **What is controllable?** N/A.

    - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the PriceAggregator contract, to which calls will be made.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `aggregator.executions()`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the Execute contract, to which calls will be made.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `executor.triggered(triggeredLimitId)`
    - **What is controllable?** `triggeredLimitId`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Checks if the limit order has been triggered and returns true if it is triggered; it is not directly controllable.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `executor.timedOut(triggeredLimitId)`
    - **What is controllable?** `triggeredLimitId`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Checks if the limit order has timed out; returns true if that is the case.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `aggregator.getPrice(_pairIndex, _orderType == LimitOrder.OPEN ? OrderType.LIMIT_OPEN : OrderType.LIMIT_CLOSE)`
    - **What is controllable?** `_pairIndex` and `_orderType`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `orderId` for the current order.
    - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert; no reentrancy scenarios.
- `this.storageT.storePendingLimitOrder(PendingLimitOrder(_trader, _pairIndex, _index, _orderType), orderId)`
    - **What is controllable?** `_trader`, `_pairIndex`, `_index`, and `_orderType`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Stores the pending limit order — no return value.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `executor.storeFirstToTrigger(triggeredLimitId, msg.sender)`
    - **What is controllable?** `triggeredLimitId` and `msg.sender`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Stores the first address to trigger the limit order — no return value.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

- `aggregator.fulfill{value: msg.value}`
  - **What is controllable?** `msg.value`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Fulfills the update margin order — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `openTrade(ITradingStorage.Trade t, IExecute.OpenLimitOrderType _type, uint256 _slippageP, bytes[] priceUpdateData, uint256 _executionFee)`

This opens a new market/limit trade.

Inputs

- `t`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The details of the trade to open.
- `_type`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Market or limit or stop limit type of trade.
- `_slippageP`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The slippage percentage.
- `priceUpdateData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Pyth price update data.
- `_executionFee`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The fee for executing the trade (in USDC).

Branches and code coverage

Intended branches

- If the order type is MARKET, store the pending market order and call the fulfill function in aggregator to fulfill the order and unregister the pending order.

  - ☑ Test coverage
- If the order type is LIMIT, store the open limit order.
  - ☑ Test coverage
- If TP and SL are provided, check if they are in correct range.
  - ☑ Test coverage

Negative behavior

- Revert if the open trades count plus the pending market open count plus the open limit-orders count is greater than or equal to the max trades per pair.
  - □ Negative test
- Revert if leverage is not in the correct range.
  - □ Negative test
- Revert if the position size multiplied by leverage is less than the minimum leverage position.
  - □ Negative test

Function call analysis

- `this.storageT.priceAggregator()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the PriceAggregator contract, to which calls will be made.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `aggregator.pairsStorage()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the TradingStorage contract, to which calls will be made.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.openTradesCount(msg.sender, t.pairIndex)`
  - **What is controllable?** `msg.sender` and `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the count of open trades for the user and trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.pendingMarketOpenCount(msg.sender, t.pairIndex)`
  - **What is controllable?** `msg.sender` and `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the count of pending market orders for the user and trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

- `this.storageT.openLimitOrdersCount(msg.sender, t.pairIndex)`
    - **What is controllable?** `msg.sender` and `t.pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the count of open limit orders for the user and trading pair.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.maxTradesPerPair()`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the max amount of trades per pair.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.pendingOrderIdsCount(msg.sender)`
    - **What is controllable?** `msg.sender`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the count of pending orders for the user.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.maxPendingMarketOrders()`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the max pending market orders.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `PositionMath.mul(t.positionSizeUSDC, t.leverage)`
    - **What is controllable?** `t.positionSizeUSDC` and `t.leverage`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Calculates the position size based on leverage.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `pairsStored.pairMinLevPosUSDC(t.pairIndex)`
    - **What is controllable?** `t.pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the minimum leverage position USDC for the trading pair.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `pairsStored.pairMinLeverage(t.pairIndex)`
    - **What is controllable?** `t.pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the minimum leverage for the trading pair.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `pairsStored.pairMaxLeverage(t.pairIndex)`

  - **What is controllable?** `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the maximum leverage for the trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.transferUSDC(msg.sender, address(this.storageT), t.positionSizeUSDC + _executionFee)`
  - **What is controllable?** `msg.sender`, `t.positionSizeUSDC`, and `_executionFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Transfers USDC from the caller to the storage contract.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.firstEmptyOpenLimitIndex(msg.sender, t.pairIndex)`
  - **What is controllable?** `msg.sender` and `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Finds the first empty open limit index for the user and trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.storeOpenLimitOrder(OpenLimitOrder(msg.sender, t.pairIndex, index, t.positionSizeUSDC, t.buy, t.leverage, t.tp, t.sl, t.openPrice, t.openPrice, block.number, _executionFee))`
  - **What is controllable?** `msg.sender`, `t.pairIndex`, `index`, `t.positionSizeUSDC`, `t.buy`, `t.leverage`, `t.tp`, `t.sl`, `t.openPrice`, `block.number`, and `_executionFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Stores an open limit order — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `aggregator.executions().setOpenLimitOrderType(msg.sender, t.pairIndex, index, _type)`
  - **What is controllable?** `msg.sender`, `t.pairIndex`, `index`, and `_type`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Sets the open limit order type — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `aggregator.getPrice(t.pairIndex, OrderType.MARKET_OPEN)`
  - **What is controllable?** `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `orderId` of the current order.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.storePendingMarketOrder(PendingMarketOrder(Trade(msg.sender, t.pairIndex, 0, 0, t.positionSizeUSDC, 0, t.buy, t.leverage, t.tp,`

`t.sl, 0), 0, t.openPrice, _slippageP), orderId, True)`
    - **What is controllable?** `msg.sender`, `t.pairIndex`, `t.positionSizeUSDC`, `t.buy`, `t.leverage`, `t.tp`, `t.sl`, `t.openPrice`, and `_slippageP`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Stores the pending market order — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `aggregator.fulfill{value: msg.value}`
    - **What is controllable?** `msg.value`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Fulfills the update margin order — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `updateMargin(uint256 _pairIndex, uint256 _index, ITradingStorage.updateType _type, uint256 _amount, bytes[] priceUpdateData)`

This updates (deposit/withdraw) the margin for an open trade.

Inputs

- `_pairIndex`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index of the trading pair for the open trade.
- `_index`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index of the open trade.
- `_type`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Withdraw or deposit type of update.
- `_amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The collateral by which to update the margin.
- `priceUpdateData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Pyth price update data.

Branches and code coverage

Intended branches

- If the trade is open and the new leverage lies between the correct range, the trade is executed.
  - ☑ Test coverage
- Tokens are transferred to the user if it is a withdraw call.
  - ☑ Test coverage
- Tokens are transferred from `trader` address to the storage contract if it is a deposit call.
  - ☑ Test coverage
- If margin fees are greater than zero, the open interest is updated.
  - ☑ Test coverage

Negative behavior

- Being market closed for the trade prevents further updates (`ALREADY_BEING_CLOSED`).
  - □ Negative test
- The trade should exist with a positive leverage.
  - □ Negative test
- The new leverage should lie between the minimum and maximum range.
  - □ Negative test

Function call analysis

- `this.storageT.priceAggregator()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the `PriceAggregator` address, to which calls will be made.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.openTrades(msg.sender, _pairIndex, _index)`
  - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade; incorrect values may lead to incorrect trade information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.openTradesInfo(msg.sender, _pairIndex, _index)`
  - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves additional information about the open trade; incorrect values may lead to incorrect information retrieval.

    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.pairInfos.getTradeRolloverFee(t.trader, t.pairIndex, t.index, t.buy, t.initialPosToken, t.leverage)`
    - **What is controllable?** `t.trader`, `t.pairIndex`, `t.index`, `t.buy`, `t.initialPosToken`, and `t.leverage`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Calculates the trade rollover fee.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `aggregator.pairsStorage().pairMinLeverage(t.pairIndex)`
    - **What is controllable?** `t.pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the minimum leverage for the trading pair.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `aggregator.pairsStorage().pairMaxLeverage(t.pairIndex)`
    - **What is controllable?** `t.pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the maximum leverage for the trading pair.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `aggregator.getPrice(_pairIndex, OrderType.UPDATE_MARGIN)`
    - **What is controllable?** `_pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the new `orderId` for the current order.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `aggregator.storePendingMarginUpdateOrder(orderId, PendingMarginUpdate(msg.sender, _pairIndex, _index, _type, _amount, i.lossProtection, marginFees, t.leverage))`
    - **What is controllable?** `orderId`, `msg.sender`, `_pairIndex`, `_index`, `_type`, `_amount`, and `t.leverage`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Stores the pending margin update order — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.updateTrade(t)`
    - **What is controllable?** `t`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Updates the trade information — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `aggregator.fulfill{value: msg.value}`

- **What is controllable?** `msg.value`.
- **If the return value is controllable, how is it used and how can it go wrong?** Fulfills the update margin order — no return value.
- **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `updateOpenLimitOrder(uint256 _pairIndex, uint256 _index, uint256 _price, uint256 _tp, uint256 _sl)`

This updates an open limit order.

Inputs

- `_pairIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the trading pair.
- `_index`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the order.
- `_price`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The price level to set (`_PRECISION`).
- `_tp`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The take-profit price.
- `_sl`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The stop-loss price.

Branches and code coverage

Intended branches

- If the new TP and SL are in the correct range, update the open limit order.
  - ☑ Test coverage

Negative behavior

- Revert if the time since the order creation is less than the defined timelock period. (Enforces timelock for order updates.)
    - □ Negative test
- Revert if `_tp` is set and not valid according to order type.
    - □ Negative test
- Revert if `_sl` is set and not valid according to order type.
    - □ Negative test

Function call analysis

- `this.storageT.getOpenLimitOrder(msg.sender, _pairIndex, _index)`
    - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the open limit order; this limit order is updated and later stored in storage.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.storageT.updateOpenLimitOrder(o)`
    - **What is controllable?** `o`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Updates the open limit order based on the provided information — no return value.
    - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `updateTpAndSl(uint256 _pairIndex, uint256 _index, uint256 _newSl, uint256 _newTP, bytes[] priceUpdateData)`

This updates the take profit and stop loss for an open trade.

Inputs

- `_pairIndex`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index of the trading pair.
- `_index`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index of the order.
- `_newSl`
    - **Control**: Fully controlled by the caller.

  - **Constraints**: None.
  - **Impact**: The new stop-loss price.
- `_newTP`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The new take-profit price.
- `priceUpdateData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Pyth price update data.

Branches and code coverage

Intended branches

- The function calls `_updateTp` and `_updateSl` to update take profit and stop loss, respectively.
- [x] Test coverage
- In `_updateSl`, if the pair does not have guaranteed stop loss enabled, call the `updateSl` in storage contract.
- [x] Test coverage
- In `_updateSl`, if the pair has guaranteed stop loss enabled, take the dev governance fees from the initial position, update the trade, and call fulfill in the aggregator to fulfill the order.
- [x] Test coverage

Negative behavior

- Revert if `block.number - tpLastUpdated` is less than `limitOrdersTimelock`.
- [] Negative test
- Revert if `block.number - slLastUpdated` is less than `limitOrdersTimelock`.
- [] Negative test
- If stop loss deviates more than `maxSlDist`, revert the transactions.
- [] Negative test
- Revert if leverage is zero.
- [] Negative test

Function call analysis

- `this._updateTp(_pairIndex, _index, _newTP) -> this.storageT.openTrades(msg.sender, _pairIndex, _index)`
  - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.

    - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade; incorrect values may lead to incorrect trade information retrieval.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._updateTp(_pairIndex, _index, _newTP) -> this.storageT.openTradesInfo(msg.sender, _pairIndex, _index)`
    - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves additional information about the open trade; incorrect values may lead to incorrect information retrieval.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._updateTp(_pairIndex, _index, _newTP) -> this.storageT.updateTp(msg.sender, _pairIndex, _index, _newTp)`
    - **What is controllable?** `msg.sender`, `_pairIndex`, `_index`, and `_newTp`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Updates the TP value — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> this.storageT.openTrades(msg.sender, _pairIndex, _index)`
    - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade; incorrect values may lead to incorrect trade information retrieval.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> this.storageT.openTradesInfo(msg.sender, _pairIndex, _index)`
    - **What is controllable?** `msg.sender`, `_pairIndex`, and `_index`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves additional information about the open trade; incorrect values may lead to incorrect information retrieval.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> this.storageT.priceAggregator()`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the PriceAggregator contract, to which calls will be made.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> aggregator.pairsStorage().guaranteedSlEnabled(_pairIndex)`
    - **What is controllable?** `_pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns if SL is guaranteed enabled for this pair index.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> aggregator.pairsStorage()`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the TradingStorage contract, to which calls will be made.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> this.storageT.updateSl(msg.sender, _pairIndex, _index, _newSl)`
    - **What is controllable?** `msg.sender`, `_pairIndex`, `_index`, and `_newSl`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Updates the SL for the order — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> this.storageT.handleDevGovFees(t.trader, t.pairIndex, levPosUSDC / 2, False, True, t.buy)`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** This call handles fees based on trade parameters, and controlling it directly is not feasible.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> this.storageT.updateTrade(t)`
    - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** Updates the trade — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> aggregator.getPrice(_pairIndex, OrderType.UPDATE_SL)`
    - **What is controllable?** `_pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `orderId` of the current order.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

- this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> aggregator.storePendingSlOrder(orderId, PendingSl(msg.sender, _pairIndex, _index, t.openPrice, t.buy, _newSl))
    - **What is controllable?** msg.sender, _pairIndex, _index, and _newSl.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- this._updateSl(_pairIndex, _index, _newSl, priceUpdateData) -> aggregator.fulfill{value: msg.value}
    - **What is controllable?** msg.value.
    - **If the return value is controllable, how is it used and how can it go wrong?** Fulfills the update margin order — no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert —- no reentrancy scenarios.

5.5. Module: Tranche.sol

Function: setWithdrawThreshold(uint256 _withdrawThreshold)

This sets the withdraw threshold for the contract.

Inputs

- _withdrawThreshold
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Should be less than 100 * _PRECISION.
    - **Impact**: The new withdraw threshold value.

Branches and code coverage

Intended branches

- Updates the value of withdrawThreshold.
    ☑ Test coverage

Negative behavior

- Revert if _withdrawThreshold is greater than 100 * _PRECISION.
    □ Negative test

Function call analysis

No external function calls found.

5.6. Module: VaultManager.sol

Function: `allocateRewards(uint256 rewards)`

This allocates rewards to the LPs.

Inputs

- `rewards`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be greater than zero.
  - **Impact**: The amount of rewards to allocate.

Branches and code coverage

Intended branches

- If the caller is not a trading contract, transfer assets from the caller to `address(this)` and increase the `totalRewards` value.
  - □ Test coverage
- If caller is a trading contract, simply increase the `totalRewards` value.
  - □ Test coverage

Negative behavior

- Revert if rewards are zero.
  - □ Negative test

Function call analysis

- `IERC20(this.junior.asset()).transferFrom(msg.sender, address(this), rewards)`
  - **What is controllable?** `msg.sender` and `rewards`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The reward amount is transferred from the caller to the contract — no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.junior.asset()`
  - **What is controllable?** N/A.

    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.7. Module: VeTranche.sol

Function: `claimRewards(uint256 tokenId)`

This claims rewards accumulated by `tokenId`.

Inputs

- `tokenId`
    - **Control**: Fully controlled by caller.
    - **Constraints**: None.
    - **Impact**: The ID of the token for which rewards are claimed.

Branches and code coverage

Intended branches

- The function internally calls `_claimRewards`, which updates the rewards and transfers rewards to the caller.
    - □ Test coverage

Negative behavior

- Revert if caller is not the owner of the `tokenId`.
    - □ Negative test

Function call analysis

- `this._claimRewards(tokenId) -> SafeERC20.safeTransfer(IERC20(this.tranche.asse this._ownerOf(tokenId), this.rewardsByTokenId[tokenId])`
    - **What is controllable?** `this._ownerOf(tokenId)` and `this.rewardsByTokenId[tokenId]`.
    - **If the return value is controllable, how is it used and how can it go wrong?** The reward amount is transferred to the owner of the token; an incorrect owner may result in incorrect reward transfer.
    - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._claimRewards(tokenId) -> this.tranche.asset()`
    - **What is controllable?** N/A.

- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `forceUnlock(uint256 _tokenId)`

This force unlocks a token if its lock time has passed — to be called by keepers.

Inputs

- `_tokenId`
  - **Control**: Fully controlled by caller.
  - **Constraints**: `lockTimeByTokenId[_tokenId]` should be less than `block.timestamp`, and `tokensByTokenId[_tokenId]` should be greater than zero.
  - **Impact**: The ID of the token to unlock.

Branches and code coverage

Intended branches

- The function internally calls `_claimRewards`, which updates the rewards and transfers rewards to the owner of `tokenId`.
  - ☑ Test coverage
- The function burns the token with `id = tokenId`.
  - ☑ Test coverage
- The shares are transferred back to the owner of the `tokenId`.
  - ☑ Test coverage

Negative behavior

- Revert if `lockTimeByTokenId[_tokenId]` > `block.timestamp`.
  - ☑ Negative test
- Revert if `tokensByTokenId[_tokenId] = 0`.
  - □ Negative test

Function call analysis

- `this._claimRewards(_tokenId) -> SafeERC20.safeTransfer(IERC20(this.tranche.ass this._ownerOf(tokenId), this.rewardsByTokenId[tokenId])`
  - **What is controllable?** `this._ownerOf(_tokenId)` and `this.rewardsByTokenId[_tokenId]`.

  - **If the return value is controllable, how is it used and how can it go wrong?** The reward amount is transferred to the owner of the token; an incorrect owner may result in incorrect reward transfer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._claimRewards(_tokenId) -> this.tranche.asset()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.tranche.transfer(this._ownerOf(_tokenId), this.tokensByTokenId[_tokenId]`
  - **What is controllable?** `this._ownerOf(_tokenId)` and `this.tokensByTokenId[_tokenId]`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The remaining unlocked tokens are transferred to the owner; incorrect values may lead to incorrect token transfer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `lock(uint256 shares, uint256 endTime)`

This locks a specified amount of shares until a specified end time.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be greater than zero.
  - **Impact**: The number of shares to lock.
- `endTime`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `endTime - block.timestamp` should be between the range `getMinLockTime()` and `getMaxLockTime()`.
  - **Impact**: The time until which the shares will be locked.

Branches and code coverage

Intended branches

- The `endTime - block.timestamp` should be between the range `getMinLockTime()` and `getMaxLockTime()`.
  - ☑ Test coverage

- Check if `balanceOf` caller is greater than the value of `shares`.
  - ☑ Test coverage
- Mint NFT to the caller and increment the `nextTokenId` counter.
  - ☑ Test coverage

Negative behavior

- The `endTime - block.timestamp` is outside the expected range.
  - □ Negative test
- The `shares` amount is equal to zero.
  - □ Negative test
- The `balanceOf` caller is less than the `shares` amount.
  - □ Negative test

Function call analysis

- `this.getMaxLockTime() -> this.vaultManager.maxLockTime()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.getMinLockTime() -> this.vaultManager.minLockTime()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.tranche.balanceOf(msg.sender)`
  - **What is controllable?** `msg.sender`.
  - **If the return value is controllable, how is it used and how can it go wrong?** It is the balance of `msg.sender` — should be greater than `shares` for a successful call.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `Counters.current(this.tokenIds)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.tranche.transferFrom(msg.sender, address(this), shares)`
  - **What is controllable?** `msg.sender` and `shares`.
  - **If the return value is controllable, how is it used and how can it go wrong?**

N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `Counters.increment(this.tokenIds)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `unlock(uint256 tokenId)`

This unlocks a specific `tokenId`.

Inputs

- `tokenId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The owner of `tokenId` should be `msg.sender`.
    - **Impact**: The ID of the token to unlock.

Branches and code coverage

Intended branches

- The number of tokens to unlock for the provided `tokenId` should be greater than zero.
    ☑ Test coverage
- The owner of `tokenId` should be `msg.sender`.
    ☑ Test coverage
- The fee returned by `checkUnlockFee` should be sent to the vault manager.
    ☑ Test coverage
- The remaining amount should be transferred to `msg.sender`, and the `tokenId` should be burned.
    ☑ Test coverage
- Updates the rewards for the caller and transfers the reward amount.
    ☑ Test coverage
- Deletes the mappings for the `tokenId`.
    ☑ Test coverage

Negative behavior

- Revert if the caller is not `msg.sender`.
    □ Negative test

- Revert if the number of tokens to unlock for the provided `tokenId` is zero.
  - □ Negative test

Function call analysis

- `this.checkUnlockFee(tokenId) -> this.getEarlyWithdrawFee(this.tokensByTokenId` `-> this.tranche.feesOn()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true if fees are on, otherwise false; return value is not in control of the user.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.checkUnlockFee(tokenId) -> this.getEarlyWithdrawFee(this.tokensByTokenId` `-> this.vaultManager.earlyWithdrawFee()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the early withdrawal fees; return value is not in control of the user.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.checkUnlockFee(tokenId) -> MathUpgradeable.mulDiv(fee, timeLeft, totalTime, Rounding.Up)`
  - **What is controllable?** `fee`, `timeLeft`, and `totalTime`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The fee calculation is based on these parameters; incorrect values may lead to incorrect fee calculation.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._claimRewards(tokenId) -> SafeERC20.safeTransfer(IERC20(this.tranche.ass` `this._ownerOf(tokenId), this.rewardsByTokenId[tokenId])`
  - **What is controllable?** `this._ownerOf(tokenId)` and `this.rewardsByTokenId[tokenId]`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The reward amount is transferred to the owner of the token; an incorrect owner will be reverted in the previous check.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this._claimRewards(tokenId) -> this.tranche.asset()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.tranche.transfer(msg.sender, this.tokensByTokenId[tokenId] - fee)`

    - **What is controllable?** `msg.sender` and `this.tokensByTokenId[tokenId] - fee`.
    - **If the return value is controllable, how is it used and how can it go wrong?** The remaining unlocked tokens are transferred to the caller; incorrect values may lead to incorrect token transfer.
    - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.tranche.transfer(address(this.vaultManager), fee)`
    - **What is controllable?** `fee`.
    - **If the return value is controllable, how is it used and how can it go wrong?** The fee amount is transferred to the vault manager; an incorrect fee may lead to incorrect fee transfer.
    - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Base mainnet.

During our assessment on the scoped Avantis contracts, we discovered 38 findings. Six critical issues were found. Seven were of high impact, eight were of medium impact, 10 were of low impact, and the remaining findings were informational in nature. Avantis Labs, Inc. acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.