avantis

SMART CONTRACTS REVIEW

zokyo

December 23rd 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING AVANTIS

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 2 High issues: 1 resolved and 1 acknowledged = -3 points deducted
- 7 Medium issues: 4 resolved and 3 acknowledged = - 6 points deducted
- 13 Low issues: = 11 resolved and 2 acknowledged = - 2 points deducted
- 19 Informational issues: 10 resolved and 9 acknowledged = -3 points deducted

Thus, 100 - 3 - 6 -2 - 3 = 86

TECHNICAL SUMMARY

This document outlines the overall security of the Avantis Labs smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Avantis Labs smart contracts codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Avantis Labs team put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Avantis Labs repository:
Repo: https://github.com/Avantis-Labs/avantis-contracts/tree/audits

Last commit -798c3a998d8a4477c7b342602def1ab708cae162

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- Execute.sol
- PairInfos.sol
- PairStorage.sol
- PriceAggregator.sol
- Referral.sol
- Trading.sol
- TradingCallbacks.sol
- TradingStorage.sol
- Tranche.sol
- VaultManager.sol
- VeTranche.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Avantis Labs smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contracts by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has not identified any critical severity issues. However, two issues with high severity have been identified, along with medium and low severity issues, and a couple of informational findings. For a more in-depth analysis of these discoveries, please consult the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Avantis Labs team and the Avantis Labs team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Tranche's withdrawal fees are not deducted properly	High	Acknowledged
2	Lack of access control in setWithdrawThreshold function	High	Resolved
3	Possibility of immediate position liquidation after contract state is unpaused	Medium	Acknowledged
4	PrincipalShareDeposited is being increased without actual transfer/minting of shares	Medium	Resolved
5	Method getEarnings() underflows	Medium	Resolved
6	Possibility of ETH being stuck in Trading contract	Medium	Acknowledged
7	Function returns wrong values	Medium	Resolved
8	Potentially losing eth	Medium	Acknowledged
9	Chainlink's latestRoundData might return stale or incorrect results	Medium	Resolved
10	Instant governance transfer	Low	Resolved
11	Function is returning the wrong first empty index	Low	Resolved
12	Tranche Transfer can potentially lead to wasted assets	Low	Resolved
13	collateralFees are initialized to values in a descending order	Low	Resolved
14	Referrer codes of user can be overridden and lost	Low	Resolved
15	ERC20 Transfers go unvalidated	Low	Resolved
16	Centralization risk in several methods	Low	Acknowledged

#	Title	Risk	Status
17	Missing _disableInitializer() implementation	Low	Resolved
18	Missing initializing PausableUpgradeable contract	Low	Resolved
19	he trading contract doesn't allow the removal of whitelisted addresses	Low	Resolved
20	MinLockTime should be less than maxLockTime	Low	Resolved
21	Method forceUnlock(...) might fail for blacklisted USDC addresses	Low	Acknowledged
22	The in-memory value won't be updated	Low	Resolved
23	Lack of PausableUpgradable initialization	Informational	Resolved
24	Insufficient validation of max lock time	Informational	Resolved
25	Unused import	Informational	Resolved
26	Trade.index is assigned twice	Informational	Acknowledged
27	Methods applyReferralOpen() and applyReferralClose() can be combined into a single method	Informational	Resolved
28	Confusing method name	Informational	Acknowledged
29	Split `require` statement with multiple conditions	Informational	Acknowledged
30	Inverse `if-else` statements that have a negation	Informational	Resolved
31	Use optimal `for loop`	Informational	Resolved
32	Right Bitshift by 1 when need to divide by 2	Informational	Resolved
33	Using inline assembly for address(0) check is cheaper	Informational	Acknowledged
34	Safemath unnecessary computation cost being incurred	Informational	Acknowledged
35	Percentage value needs to be properly bounded	Informational	Acknowledged
36	Redundant operation	Informational	Acknowledged

#	Title	Risk	Status
37	Redundant if statement	Informational	Resolved
38	Misleading revert messages	Informational	Acknowledged
39	Vague revert messages	Informational	Acknowledged
40	Repeated instruction	Informational	Resolved
41	Pair delisted but not deleted	Informational	Resolved

Tranche's withdrawal fees are not deducted properly

In Contract Tranche.sol, methods redeem/withdraw calls _withdraw(...) method internally. The method _withdraw has the following logic:

```
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);

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

Here, the fee is calculated but the whole amount of `assets` is sent to the receiver address without the fee deduction.
Instead, it is sent from the vault to the vault manager in the next line which is an extra amount being paid as a fee.

PoC:

```
function testWithdrawalFee() public {
    uint rand = uint(keccak256(abi.encodePacked(block.timestamp))) %
        numTraders;
    address ownerTrader = traders[rand];
    vm.startPrank(ownerTrader);
    uint amountDeposited = usdc.balanceOf(ownerTrader);
    usdc.approve(address(juniorTranche), amountDeposited);

    emit log_named_uint("usdc amount deposited", amountDeposited);
    juniorTranche.deposit(amountDeposited, ownerTrader);

    emit log_named_uint(
        "minted shares",
        juniorTranche.balanceOf(ownerTrader)
    );

    uint withdrawalFeeRaw = juniorTranche.getWithdrawalFeesRaw(
        juniorTranche.balanceOf(ownerTrader)
    );
    emit log_named_uint("withdraw fee raw", withdrawalFeeRaw);

    emit log_named_uint(
        "withdraw fee total",
        juniorTranche.getWithdrawalFeesTotal(
            juniorTranche.balanceOf(ownerTrader)
        )
    );

    uint redeemB = juniorTranche.redeem(
        juniorTranche.balanceOf(ownerTrader),
        ownerTrader,
        ownerTrader
    );
```

```
        emit log_named_uint("redeem return assets", redeemB);

        emit log_named_uint(
            "usdc balance after redeem for owner",
            usdc.balanceOf(ownerTrader)
        );

        emit log_named_uint(
            "lp shares for owner",
            juniorTranche.balanceOf(ownerTrader)
        );

        assertEq(
            usdc.balanceOf(ownerTrader),
            amountDeposited - withdrawalFeeRaw
        );
    }
```

This test returns the following log:

```
❯ forge test --match-contract Vault --match-test testWithdrawalFee -vv
[:] Compiling...
No files changed, compilation skipped

Running 1 test for test/units/Vault.t.sol:Vault
[FAIL. Reason: Assertion failed.] testWithdrawalFee() (gas: 324639)
Logs:
  usdc amount deposited: 5743269554
  minted shares: 5743269554
  withdraw fee raw: 1435818
  withdraw fee total: 1435459
  redeem return assets: 5741834095
  usdc balance after redeem for owner: 5741834095
  lp shares for owner: 0
  Error: a == b not satisfied [uint]
        Left: 5741834095
       Right: 5741833736

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 8.32ms

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/units/Vault.t.sol:Vault
[FAIL. Reason: Assertion failed.] testWithdrawalFee() (gas: 324639)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommendation:
update the logic to send `assets-fee` to the receiver during withdrawal.

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

Fix: The client acknowledged the issue with the comment “While it looks like the fee is not being deducted, the fee will be shown on the UI as something the user has to pay on top of the assets they deposit”.

HIGH-2 | RESOLVED

Lack of access control in setWithdrawThreshold function

The setWithdrawThreshold function from Tranche contract is responsible for setting a threshold that dictates the conditions under which withdrawals can occur from the contract. However, the function can be invoked by any external actor without restriction. A malicious actor could set the withdrawal threshold to a zero value, effectively freezing the contract's funds by making the utilization ratio limit unachievable. Setting it to zero would mean users could never meet the withdrawal criteria, effectively preventing all users from retrieving their assets.

Recommendation:
Use an onlyGov modifier similar to that used for other governance functions.

MEDIUM-1 | ACKNOWLEDGED

Possibility of immediate position liquidation after contract state is unpaused

When the updateMargin function within Trading contract is paused, traders are unable to adjust their margins to respond to market volatility. If the market moves against a trader's position during the pause, they could fall below the required maintenance margin without any recourse to rectify the situation by depositing additional collateral. Consequently, once the contract resumes and updateMargin is re-enabled, these under-margined positions become immediate targets for liquidation bots.

Recommendation:
Consider introducing a grace period after resuming contract functions to allow traders to update their margins before any liquidations can be processed.

PrincipalShareDeposited is being increased without actual transfer/minting of shares

In Contract Tranche.sol, the method _withdraw(...) has the following logic:

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

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

      // 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);
      }
  }
```

Here, when receiver != owner, the mapping principalShareDeposited and principalAssetsDeposited are being increased for the receiver address after the shares of the owner have been burned and assets have been sent to the receiver address.

This is confusing since there are no shares/LP tokens transferred to the receiver address but according to the mapping principalShareDeposited, the receiver owns some shares.

This is also contrary to the _deposit() method where actual shares are minted and also in _transfer() where shares are transferred from owner to receiver.
PoC:

```
function testReedem() public {
    uint rand = uint(keccak256(abi.encodePacked(block.timestamp))) %
        numTraders;
    address ownerTrader = traders[rand];
    vm.startPrank(ownerTrader);
    uint amount = usdc.balanceOf(ownerTrader);
    usdc.approve(address(juniorTranche), amount);

    emit log_named_uint("usdc amount deposited", amount);
    juniorTranche.deposit(amount, ownerTrader);

    emit log_named_uint(
        "minted shares",
        juniorTranche.balanceOf(ownerTrader)
    );

    address receiverTrader = makeAddr("receiver");
    uint redeemB = juniorTranche.redeem(
        juniorTranche.balanceOf(ownerTrader),
        receiverTrader,
        ownerTrader
    );

    emit log_named_uint("redeem return assets", redeemB);

    emit log_named_uint(
        "usdc balance after redeem for owner",
        usdc.balanceOf(ownerTrader)
    );
```

```
emit log_named_uint(
    "lp shares for owner",
    juniorTranche.balanceOf(ownerTrader)
);

emit log_named_uint(
    "asset deposited  for owner",
    juniorTranche.principalAssetsDeposited((ownerTrader))
);

emit log_named_uint(
    "shares deposited for owner",
    juniorTranche.principalSharesDeposited((ownerTrader))
);

emit log_named_uint(
    "usdc balance after redeem for receiver",
    usdc.balanceOf(receiverTrader)
);

emit log_named_uint(
    "lp shares for receiver",
    (juniorTranche.balanceOf(receiverTrader))
);

emit log_named_uint(
    "asset deposited  for receiver",
    juniorTranche.principalAssetsDeposited((receiverTrader))
);

emit log_named_uint(
    "shares deposited for receiver",
    juniorTranche.principalSharesDeposited((receiverTrader))
);

}
```

This test logs the following result:

```
Logs:
  usdc amount deposited: 5743269554
  minted shares: 5743269554
  redeem return assets: 5741834095
  usdc balance after redeem for owner: 0
  lp shares for owner: 0
  asset deposited  for owner: 0
  shares deposited for owner: 0
  usdc balance after redeem for receiver: 5741834095
  lp shares for receiver: 0
  asset deposited  for receiver: 5740398636000000000
  shares deposited for receiver: 5743269554
```

Here, we can see the receiver LP shares are `0` but shares deposited for the receiver are > `0`. It gives the false implication that the receiver has shares that can be withdrawn for assets but it will fail.

Recommendation:
Update the _withdraw logic to avoid a misleading scenario to maintain consistency between real shares balance and mapping principalSharesDeposited.

Fix: Issue fixed, It is advised to remove the unused mappings principalAssetsDeposited & principalSharesDeposited.

Method getEarnings() underflows

As we notice in the above findings, the principalAssetDeposited is being increased for a receiver who has 0 LP tokens or no shares. The method getEarning(...) returns an unexpected result if it is checked for the same receiver.

PoC:

```
function testGetEarnings() public {
    uint rand = uint(keccak256(abi.encodePacked(block.timestamp))) %
        numTraders;
    address ownerTrader = traders[rand];
    vm.startPrank(ownerTrader);
    uint amount = usdc.balanceOf(ownerTrader);
    usdc.approve(address(juniorTranche), amount);

    emit log_named_uint("usdc amount deposited", amount);
    juniorTranche.deposit(amount, ownerTrader);

    emit log_named_uint(
        "minted shares",
        juniorTranche.balanceOf(ownerTrader)
    );
    address receiverTrader = makeAddr("receiver");

    emit log_named_uint(
        "earning for receiver before redeem",
        uint256(juniorTranche.getEarnings(receiverTrader))
    );
    uint redeemB = juniorTranche.redeem(
        juniorTranche.balanceOf(ownerTrader),
        receiverTrader,
        ownerTrader
    );

    emit log_named_uint("redeem return assets", redeemB);
    emit log_named_uint(
        "usdc balance after redeem for receiver",
        usdc.balanceOf(receiverTrader)
    );

    emit log_named_uint(
        "lp shares for receiver",
        (juniorTranche.balanceOf(receiverTrader))
    );
    emit log_named_uint(
        "earning for receiver after redeem",
```

```
            uint256(juniorTranche.getEarnings(receiverTrader))
        );
    }
```

This test logs the following results:

```
Logs:
  usdc amount deposited: 5743269554
  minted shares: 5743269554
  earning for receiver before redeem: 0
  redeem return assets: 5741834005
  usdc balance after redeem for receiver: 5741834005
  lp shares for receiver: 0
  earning for receiver after redeem: 115792089237316195423570985008687907853269984665640564039457584007913129639936
```

Here, the earnings for the receiver after redemption is a very large value which is clearly due to underflow.

Recommendation:
Update the _withdraw logic to avoid a misleading scenario to maintain consistency between real shares balance and mapping principalAssetsDeposited.

Possibility of ETH being stuck in Trading contract

In Contract Trading.sol, the methods updateSl(...) and updateTpAndSl(...) allows users to update the SL of an open trade. These methods internally call the _updateSl(...) method which has the following check.

```
        if (
            _newSl == 0 ||
            !aggregator.pairsStorage().guaranteedSlEnabled(_pairIndex)
        ) {
            storageT.updateSl(msg.sender, _pairIndex, _index, _newSl);
            emit SlUpdated(
                msg.sender,
                _pairIndex,
                _index,
                _newSl,
                block.timestamp
            );
        } else {
            .
            .
            aggregator.fulfill{value: msg.value}(orderId,
priceUpdateData);
        }
```

In this logic, `msg.value` passed as a fee for the fulfill(...) method is used only in the `else` branch. Otherwise, it is just kept in the contract and never used.

Recommendation:

Add a method to withdraw the stuck ETH in case. Also, check off-chain if the contract has ETH already for the oracle fee and send accordingly.

MEDIUM-5 | RESOLVED

Function returns wrong values

Referral.sol - Function `traderReferralDiscount()` returns significant false values of `traderFeesPostDiscount` and `rebateShare`.

That takes place in a scenario where `tiers[_tierId].feeDiscountPct` is zero. In that `case traderFeesPostDiscount` should be equal to `_fee` because there is zero discount. And so `rebateShare` should be non-zero since it is derived from `traderFeesPostDiscount`.

Recommendation:
Return the correct values in the case of zero `tiers[_tierId].feeDiscountPct`.

Fix: Issue was addressed successfullthe y by client in dad4a2c6e161294019e0fd1ea7a89e797647c2e5 by removing the if statement.

MEDIUM-6 | ACKNOWLEDGED

Potentially losing eth

Trading.sol - Function `openTrade` accepts native coin (i.e. ETH in Ethereum main net) as it is `payable` function. Users interacting with this function might send native coin by mistake while the the `_type` is not equal to `IExecute.OpenLimitOrderType.MARKET`. Therefore the contract is expecting USDC instead of native coin. In that scenario the coin sent is being wasted in the contract and not returned to the caller.

Recommendation:
Revert if msg.value is non-zero for the unexpected `_type`.

Fix: While issue persists, it is acknowledged by client. It becomes a design choice which aims at saving the gas cost that comes with the required extra checks. That extra gas cost would be incurred by traders in most cases in order to protect careless traders in rare occasions, hence the choice to keep that code as it is.

MEDIUM-7 | RESOLVED

Chainlink's latestRoundData might return stale or incorrect results

The fulfill function in the PriceAggregator contract use Chainlink oracle as a second oracle to retrieve price data by calling latestRoundData. However, there's a risk that this data may be stale or incorrect due to various reasons related to Chainlink oracles. There is no check if the return value indicates stale data. This could lead to stale prices according to the Chainlink documentation:

- https://docs.chain.link/docs/historical-price-data/#historical-rounds

Recommendation:
Check the updatedAt parameter returned from latestRoundData() and compare it to a staleness threshold.

LOW-1 | RESOLVED

Instant governance transfer

Contracts TradingStorage.sol, Referral.sol, Vester.sol, VaultManager.sol use setGov function for transferring ownership. In case of a mistake in the provided address, the management of the particular contract will be irretrievably lost.

Recommendation:
Modify the process of updating the governance to be a two-step process. This will require the new owner to explicitly accept the ownership update.

Function is returning the wrong first empty index

TradingStorage.sol - Function `firstEmptyTradeIndex()` mistakenly returns index = `0` since this is the default value to be returned. This takes place when the `_openTrades` array is filled as the loop surpasses `maxTradesPerPair` iterations and spits out the default return value zero.

```
    function firstEmptyTradeIndex(address trader, uint pairIndex) public
view override returns (uint index) {
        for (uint i = 0; i < maxTradesPerPair; i++) {
            if (_openTrades[trader][pairIndex][i].leverage == 0) {
                index = i;
                break;
            }
        }
    }

It also takes place here:
    function firstEmptyOpenLimitIndex(address trader, uint pairIndex)
public view override returns (uint index) {
        for (uint i = 0; i < maxTradesPerPair; i++) {
            if (!hasOpenLimitOrder(trader, pairIndex, i)) {
                index = i;
                break;
            }
        }
    }
```

The severity of the bug could have been more serious if `Trading.openTrade()` does not validate the count of trades being executed by the trader. Fortunately, there is a check to prevent that in `Trading.openTrade()` which overcomes that bug:

```
252     require(
253         storageT.openTradesCount(msg.sender, t.pairIndex) +
254             storageT.pendingMarketOpenCount(msg.sender, t.pairIndex)
+
255             storageT.openLimitOrdersCount(msg.sender, t.pairIndex) <
256             storageT.maxTradesPerPair(),
257         "MAX_TRADES_PER_PAIR"
258     );
```

This bug though is having affecting `Multicall.getFirstEmptyTradeIndexes()` since it also would return false zeroes.

Recommendation:
Function better reverts if there is no empty index (i.e. runover the for loop).

LOW-3 | RESOLVED

Tranche Transfer can potentially lead to wasted assets

Tranche.sol - function `_transfer()` does not assert that `sender` and `recipient` are not the same address. While this is not an issue in standard ERC20 since it does not leave side effects. It poses a potential issue in this case since the transfer to same address leaves a side effect due to `_updateNegativePrincipal()`.

Recommendation:
Transferring to the same address needs to be disallowed.

Fix: Due to a big change in the implementation of the function, the issue becomes no longer relevant.

collateralFees are initialized to values in a descending order

VaultManager.sol - `collateralFees` is initialized to an array of numbers that are arranged in a descending order.

```
collateralFees = [250, 150, 100, 25, 10];
```

Despite that `setCollateralFees` implementation requires that elements of array be arranged in an ascending order as shown here:

```
require(_collateralFees[i] < _collateralFees[i + 1],
"NOT_DESCENDING_ORDER");
```

Recommendation:
Rearrange the values of `collateralFees` that are assigned on initialization.

Referrer codes of user can be overridden and lost

Referral.sol - In `registerCode()` the mapping `codes[msg.sender]` can be overridden by a new _code on each time the sender invokes this function.

```
  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);
    }
```

The issue arises in functions `setCodeOwner()`, `govSetCodeOwner()` as well. It overrides any code that might be already registered by the address of `_newAccount`.

```
   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);
    }

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

```
        address account = codeOwners[_code];
        delete codes[account];

        codeOwners[_code] = _newAccount;
        codes[_newAccount] = _code;

        emit GovSetCodeOwner(_code, _newAccount);
    }
```

Recommendation:
Validate that the new account is not affiliated with any code before the transfer of code ownership takes place.

Fix: The issue still persists, with a change of which function contains the issue:
setCodeOwner() → setPendingCodeOwnershipTransfer()
govSetCodeOwner unchanged

LOW-6 | RESOLVED

ERC20 Transfers go unvalidated

In TradingStorage.sol - function `transferUSDC()` does not validate the returned bools of the ERC20 transfers. So in `handleDevGovFees, claimFees` and `claimRebate.`
Also in VaultManager.sol - function `allocateRewards()`:
`IERC20(junior.asset()).transferFrom(msg.sender, address(this), rewards)`

as well as in function `sendReferrerRebateToStorage()`
`IERC20(junior.asset()).transfer(address(storageT), _amount);`

as well as in function `_distributeVeRewards()`
`IERC20(junior.asset()).transfer(address(veTranche), rewards);`

as well as in function `_distributeRewards()`
`IERC20(junior.asset()).transfer(tranche, rewards);`

Recommendation:
Use SafeERC20

LOW-7 | ACKNOWLEDGED

Centralization risk in several methods

Across the protocol, several methods use the modifier onlyGov() which can be used to configure important parameters for the protocol such as Pyth oracles, backup oracles, etc.. But in the TradingStorage contract, gov is being set to the deployer address which is an EOA.

This risks the whole protocol being centralized and controlled by a single EOA.

Recommendation:
It is advised to decentralize the usage of these functions by using a multisig wallet with at least 2/3 or a 3/5 configuration of trusted users. Alternatively, a secure governance mechanism can be utilized for the same.

LOW-8 | RESOLVED

Missing _disableInitializer() implementation

The following contracts inherit the Intializable.sol and implement the initialize(...) method with the initializer modifier without disabling the initializers for the implementation contract as recommended by OpenZeppelin here.

Execute.sol
PairInfos.sol
PairStorage.sol
PriceAggregator.sol
Trading.sol
TradingCallbacks.sol
TradingStorage.sol
Tranche.sol
VaultManager.sol
VeTranche.sol

Recommendation:
Disable the initializers for the implementation method as suggested by OpenZeppelin here.

LOW-9 | RESOLVED

Missing initializing PausableUpgradeable contract

In Contract Trading.sol, PausableUpgradeable is inherited but not initialized.

Recommendation:
Initialize the PausableUpgradeable in the initialize() method adding __Pausable_init().

LOW-10 | RESOLVED

The trading contract doesn't allow the removal of whitelisted addresses

In Contract Trading.sol, method addWhitelist(...) allows gov to whitelist any address for accessing all trading methods. But there is no method to remove this whitelisted address if it's needed in the future or if any whitelisted address turns malicious.

Recommendation:
Add a method to allow removing whitelisted addresses in the trading contract.

LOW-11 | RESOLVED

MinLockTime should be less than maxLockTime

In Contract VaultManager.sol, the setMinLockTime(...) assigns a value to minLockTime but it is not checked to be less than maxLockTime. In case minLockTime is accidentally set to greater than maxLockTime, unexpected issues might be encountered.

Recommendation:
Add a check to ensure minLockTime < maxLockTime.

LOW-12 | ACKNOWLEDGED

Method forceUnlock(...) might fail for blacklisted USDC addresses

In Contract VeTranche.sol, method lock() allows anyone to lock their LP tokens for a time period and get an NFT as a token receipt.
In case users do not unlock their tokens after lock period is over, the Platform can force unlock by burning the NFT token and sending the locked LP tokens along with a reward paid in USDC.
Any malicious user can deposit a minimal amount of USDC to mint LP tokens and lock those tokens to get an NFT token and pass that NFT token to a blacklisted USDC address.

In case of a blacklisted address in the USDC contract, this sending of USDC rewards will fail and forceUnlock(...) will fail to unlock the LP tokens probably locked forever unless the malicious user sends the NFT to another address.

Recommendation:
Add a try/catch for the USDC rewards transfer in the claimRewards() method.

LOW-13 | RESOLVED

The in-memory value won't be updated

The function updateSlCallback in the TradingCallback contract is intended to update a stop loss for a trading order. The issue here is that t is a memory variable, which means it only exists within the scope of the function call and is not stored on the blockchain. Therefore, updating t.timestamp has no effect on the persistent data. This line of code will only change the timestamp of the local copy of the trade data, not the version that is stored on-chain in storageT.

Recommendation:
The function should modify a state variable (stored in storage) instead of a memory variable.

INFORMATIONAL-1 | RESOLVED

Lack of PausableUpgradable initialization

The Trading contract derives from PausableUpgradeable, however, it fails to initialize the inherited features.

Recommendation:
Initialize the PausableUpgradeable contract.

INFORMATIONAL-2 | RESOLVED

Insufficient validation of max lock time

The function setMaxLockTime in VaultManager is designed to set the maximum lock time for locking shares in the VeTranche contract. However, there is a lack of a check to ensure that the new maximum lock time (_maxLockTime) is always greater than the already defined minimum lock time (minLockTime).

Recommendation:
Check if the new lock time is greater than minLockTime.

INFORMATIONAL-3 | RESOLVED

Unused import

In Contract TradingCallbacks, PausableUpgradeble is imported but never used.

In Contract Tranche.sol, ReentrancyGuardUpgradeable is imported, inherited, and initialized but never used.

Recommendation:
Remove unused imports.

Trade.index is assigned twice

In Contract TradingStorage, the method storeTrade(...) assigns _trade.index the return of method firstEmptyTradeIndex(...).

Although, storeTrade() is being called by the method _registerTrade(...) in the Contract TradingCallbacks. The method _registerTrade() is also assigning trade.index using the same method firstEmptyTradeIndex(...).

The same applies to the method firstEmptyOpenLimitIndex(...) in Trading.sol and TradingStorage.sol contracts.

Recommendation:
It is advised to check if the assigned trade.index passed to the storeTrade(...) method correct or not instead of reassigning it. Apply the same for firstEmptyOpenLimitIndex(...).

Methods applyReferralOpen() and applyReferralClose() can be combined into a single method

In Contract TradingStorage, the methods applyReferralOpen() and applyReferralClose() have the same logic with the only exception being the return parameters and their values. These methods can be combined as follows:

```
function applyReferral(
      bool open,
      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
          );
          if(open) {
                return (traderFeePostDiscount - referrerRebate,0);
          }
          return (traderFeePostDiscount, referrerRebate);
       }
    if(open) {
      return (_fees,0);
    }
    return (_fees, referrerRebate);
  }
```

Recommendation:

Use the above method to combine applyReferralOpen() and applyReferralClose().

INFORMATIONAL-6 | ACKNOWLEDGED

Confusing method name

In Contract VaultManager, the method _receiveUSDCFromTrader(...) is actually transferring the TradeStorage contract to the VaultManager contract but this method specifies it is from the trader. It is noted that the trader funds are first transferred to the storage contract and then to the vault manager but adding a comment to clarify it can be better.

Recommendation:
Add a comment to explain why funds are being transferred from the storage contract rather than the trader as the method name mentions.

INFORMATIONAL-7 | ACKNOWLEDGED

Split `require` statement with multiple conditions

In Contract PairInfos, the method setPercentDepthArray(..) has a `require` statement which has multiple conditions.

In Contract PairStorage, modifier groupOk(...) and feeOk(...) have a `require` statement which has multiple conditions.

In Contract PriceAggregator, the method fullfill(...) has a `require` statement to check the price and conf.

In Contract Trading, several methods have `require` statements with multiple conditions.

In Contract TradingCallbacks, method setFeeP has a `require` statement with multiple conditions.

Recommendation:
Split the conditions in the `require` statements.

INFORMATIONAL-8 | RESOLVED

Inverse `if-else` statements that have a negation

In contract PairInfo, the method lossProtectionTier(...) has a `if-else` statement

```
if (!_trade.buy) {
   action1
} else {
   action2
}
```

Can be switched to save gas.

```
if (_trade.buy) {
   action2
} else {
   action1
}
```

INFORMATIONAL-9 | RESOLVED

Use optimal `for loop`

Across the protocol, use the for loop in an optimal way as follows:

```
for (uint256 i; i < limit; ) {

    // inside the loop

    unchecked {
        ++i;
    }
}
```

Also, in the Vault Manager contract, `for` loop `++i` can be `unchecked { ++i;}

INFORMATIONAL-10 | RESOLVED

Right Bitshift by 1 when need to divide by 2

Across the protocol, there are occurrences of value being divided by 2. It can be made cheaper by right-shifting them by 1.

In Trading.sol, levPosUSDC/2 can be levPosUSDC >> 1.

In PriceAggregator, the method _median can be updated as follows:

```
function _median(uint[] memory array) public pure returns (uint) {
        _sort(array, 0, array.length);
        return
            array.length & uint256(1) == 0
                ? ((array[(array.length >> 1) - 1] +
                    array[array.length >> 1]) >> 1)
                : array[array.length >> 1];
    }
```

INFORMATIONAL-11 | ACKNOWLEDGED

Using inline assembly for address(0) check is cheaper

Across the protocol, there are many parameters being checked if they are address(0) or not. To reduce gas costs, we can use the following assembly code format.

```
function checkOptimized(address _caller) public pure returns (bool) {
        assembly {
            if iszero(_caller) {
                mstore(0x00, 0x20)
                mstore(0x20, 0x0c)
                mstore(0x40,
0x5a65726f20416464726573730000000000000000000000000000000000000000) //
load hex of "Zero Address" to memory
                revert(0x00, 0x60)
            }
        }
}
```

Safemath unnecessary computation cost being incurred

VaultManager.sol - i++ does not need to be safely computed.

```
function setCollateralFees(uint256[5] memory _collateralFees) external
onlyGov {
        for (uint i = 0; i < _collateralFees.length; ) {
            require(_collateralFees[i] < 10000, "TOO_HIGH");
            if (i != _collateralFees.length - 1)
                require(_collateralFees[i] < _collateralFees[i + 1],
"NOT_DESCENDING_ORDER");
            i++;
        }
        collateralFees = _collateralFees;
    }

    function setBufferThresholds(uint256[5] calldata _bufferThresholds)
external onlyGov {
        for (uint i; i < _bufferThresholds.length; ) {
            if (i != _bufferThresholds.length - 1)
                require(_bufferThresholds[i] < _bufferThresholds[i + 1],
"NOT_DESCENDING_ORDER");
            i++;
        }
        bufferThresholds = _bufferThresholds;
    }
```

Also there is `totalRewards -= _amount;` despite it is asserted before that `totalRewards >= _amount`.

```
   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);
    }
```

In Tranche.sol - totalReserved += amount; and totalReserved -= amount; can be unchecked safely.

```
   /**
     * @notice Reserve a specific amount of balance.
     * @param amount The amount to reserve.
     */
    function _reserveBalance(uint256 amount) internal {
        require(super.totalAssets() >= amount + totalReserved,
"RESERVE_AMOUNT_EXCEEDS_AVAILABLE");
        totalReserved += amount;
        emit BalanceReserved(amount);
    }

    /**
     * @notice Release a specific amount of reserved balance.
     * @param amount The amount to release from the reserve.
     */
    function _releaseBalance(uint256 amount) internal {
        require(totalReserved >= amount,
"RELEASE_AMOUNT_EXCEEDS_AVAILABLE");
        totalReserved -= amount;
        emit BalanceReleased(amount);
    }
```

Recommendation:
Wrap operations within unchecked as they are ensured priorly they are within the safe bounds.

Fix: Despite that client carried out useful gas savings, the specific examples from the codebase mentioned here are not addressed but acknowledged.

INFORMATIONAL-13 | ACKNOWLEDGED

Percentage value needs to be properly bounded

PairStorage.sol - In function `updateLossProtectionMultiplier()` , caller feeds the function by array `_multiplierPercent` and it is not being checked to be less than 100.

```
222     require(_multiplierPercent[i] >= _MAX_LOSS_REBATE,
"REBATE_EXCEEDS_MAX");
```

Recommendation:
add require (_multiplierPercent[i] < 100).

INFORMATIONAL-14 | ACKNOWLEDGED

Redundant operation

Trading.sol - Function openTrade on line 277 calls `firstEmptyOpenLimitIndex()` in order to get the first empty index to open an order. But it is worth noting that `storeOpenLimitOrder()` already carries out that operation in its implementation.

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

            storageT.storeOpenLimitOrder(
                ITradingStorage.OpenLimitOrder(
                    msg.sender,
                    t.pairIndex,
283                 index,

222     require(_multiplierPercent[i] >= _MAX_LOSS_REBATE,
"REBATE_EXCEEDS_MAX");
```

Recommendation:
Omit line 277 since it is already being executed in the implementation of `storeOpenLimitOrder()`.

Redundant if statement

TradingCallbacks.sol - In function `executeLimitCloseOrderCallback()`, the condition `v.reward > 0` is already being checked in the outer if-statement line 380.

```
380         if (o.orderType != ITradingStorage.LimitOrder.LIQ && v.reward
> 0) {
381             uint usdcSentToTrader = _unregisterTrade(
382                 t,
383                 v.profitP,
384                 v.posUSDC,
385                 v.reward,
386                 (v.posToken.mul(t.leverage) *
aggregator.pairsStorage().pairCloseFeeP(t.pairIndex)) /
387                     100 /
388                     _PRECISION,
389                 i.lossProtection
390             );

392             if (v.reward > 0) {
                    executor.distributeReward(
                        IExecute.TriggeredLimitId(o.trader, o.pairIndex,
o.index, o.orderType),
                        v.reward
                    );
```

Recommendation:

o need to have the if-statement at line 392.

Misleading revert messages

VaultManager.sol - Functions `setCollateralFees()` and `setBufferThresholds()` show a misleading revert message. For instance the elements of `_collateralFees` array are supposed to go in an ascending order (i.e. inceasing in value). But the revert message presents `NOT_DESCENDING_ORDER` which implies that the array should be put in descending order.

```
function setCollateralFees(uint256[5] memory _collateralFees) external
onlyGov {
        for (uint i = 0; i < _collateralFees.length; ) {
            require(_collateralFees[i] < 10000, "TOO_HIGH");
            if (i != _collateralFees.length - 1)
                require(_collateralFees[i] < _collateralFees[i + 1],
"NOT_DESCENDING_ORDER");
            i++;
        }
        collateralFees = _collateralFees;
    }

    function setBufferThresholds(uint256[5] calldata _bufferThresholds)
external onlyGov {
        for (uint i; i < _bufferThresholds.length; ) {
            if (i != _bufferThresholds.length - 1)
                require(_bufferThresholds[i] < _bufferThresholds[i + 1],
"NOT_DESCENDING_ORDER");
            i++;
        }
        bufferThresholds = _bufferThresholds;
    }
```

Recommendation:

Correct the misleading revert message.

Vague revert messages

PriceAggregator.sol - In function `fulfill()`, there are two cases in which the transaction fails. If the backup price (extracted by backup feed) is not within the accepted bounds of the price, the transaction fails. The two cases resembles whether back price is greater than or less than the price. It can be more helpful if the revert message shows which of the two cases that this transaction has failed.

```
if (bkPrice > price) {
                        require(
                            (((bkPrice - price) * 100 * _PRECISION) /
price) <= backupFeed.maxDeviationP,
                            "BACKUP_DEVIATION_TOO_HIGH"
                        );
                    }
                    if (bkPrice < price) {
                        require(
                            (((price - bkPrice) * 100 * _PRECISION) /
bkPrice) <= backupFeed.maxDeviationP,
                            "BACKUP_DEVIATION_TOO_HIGH"
                        );
                    }
```

Recommendation:

Edit the revert message so that it gives information to the reader whether the failure came in a state of `bkPrice < price or bkPrice > price` as it can be helpful.

INFORMATIONAL-18 | RESOLVED

Repeated instruction

VeTranche.sol - In functions `unlock()` and `forceUnlock()`, we have `lockStartTimeByTokenId[tokenId]` gets to be deleted twice.

```
        delete tokensByTokenId[tokenId];
         delete rewardsByTokenId[tokenId];
         delete lockTimeByTokenId[tokenId];
         delete lockStartTimeByTokenId[tokenId];
         delete lockMultiplierByTokenId[tokenId];
         delete lockStartTimeByTokenId[tokenId];
```

Recommendation:
no need to repeat `delete lockStartTimeByTokenId[tokenId]`

INFORMATIONAL-19 | RESOLVED

Pair delisted but not deleted

PairStorage.sol - `pairs[_pairIndex]` is not deleted despite that it is effectively dereferenced here. Deleting that record should be useful to release storage being used.

```
  function delistPair(
        uint _pairIndex
    ) external onlyGov {
        Pair storage p = pairs[_pairIndex];
        require(isPairListed[p.from][p.to], "PAIR_NOT_LISTED");

        isPairListed[p.from][p.to]= false;
        emit PairUpdated(_pairIndex);
    }
```

Recommendation:
delete pairs[_pairIndex]

	Execute.sol PairInfos.sol PairStorage.sol PriceAggregator.sol Referral.sol Trading.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	TradingCallbacks.sol TradingStorage.sol Tranche.sol VaultManager.sol VeTranche.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Avantis Labs team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Avantis Labs team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo