Zellic

Prepared for
Brank Dev
Avantis Labs, Inc.

Prepared by
Nipun Gupta
Jisub Kim
Zellic

September 11, 2024

Avantis
Smart Contract Security Assessment

Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 6

2. Introduction **6**

2.1. About Avantis 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Liquidations and SL/TP trigger might fail due to unnecessary check 11

3.2. More rewards allocated to the vault than available 13

3.3. Referrer rebate not paid by traders while closing a position 15

3.4. Incorrect type verification in force unregistration 17

3.5. Require check in `distributePnlRewardsFraction` could be improved 19

3.6. Pending `codeOwners` are not deleted in `govSetCodeOwner` 20

3.7. Invalid pair index 22

3.8. PairInfos could not update the address of PairStorage 23

3.9. Initialize functions are front-runnable 24

4. Discussion 24

4.1. Referral contract inherits PausableUpgradeable, but it is not used 25

4.2. Broken checks-effects-interactions patterns 25

4.3. Ambiguous comments 26

5. Threat Model 26

5.1. Module: Trading.sol 27

6. Assessment Results 37

6.1. Disclaimer 38

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Avantis Labs, Inc. from August 26th to September 5th, 2024. During this engagement, Zellic reviewed Avantis's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- Are the liquidations triggered at correct prices?
- Are there instances where values for traders and liquidity providers are inaccurate?
- Are there cases where users have more access than intended, potentially allowing them to run bots for executing their own trades?
- Are there oracle risk or configuration bugs with delegations?
- Are the new Pnl type of trades being executed as expected?

1.3. 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 orders, and stop-limit orders
- Bots responsible for unlocking overdue locked tranches, distributing rewards at regular intervals, setting order-book 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.

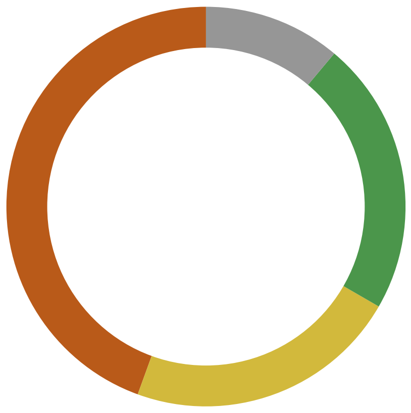
1.4. Results

During our assessment on the scoped Avantis contracts, we discovered nine findings. No critical issues were found. Four findings were of high impact, two were of medium impact, two were of low impact, and the remaining finding was informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	4
🟨 Medium	2
🟩 Low	2
⬜ Informational	1



2. Introduction

2.1. About Avantis

Avantis Labs, Inc. contributed the following description of 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 trade-able assets and high leverage (up to 100x). We are also unlocking fine-grained risk management for our 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

Type	Solidity
Platform	EVM-compatible

Target	avantis-contracts
Repository	https://github.com/brankdev/avantis-contracts ↗
Version	`305e3d9b97b7c91fb491fc36276a72e78e7348e0`

Programs	`PairInfos.sol`
	`PairStorage.sol`
	`PriceAggregator.sol`
	`Referral.sol`
	`Trading.sol`
	`TradingCallbacks.sol`
	`TradingStorage.sol`
	`Tranche.sol`
	`VaultManager.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.5 person-weeks. The assessment was conducted by two consultants over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io ↗

Jisub Kim
Engineer
jisub@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

August 26, 2024	Start of primary review period
August 28, 2024	Kick-off call
September 5, 2024	End of primary review period

3. Detailed Findings

3.1. Liquidations and SL/TP trigger might fail due to unnecessary check

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

Description

When a trade is unregistered, a part of the trader fee goes to the referrer as the `referrerRebate`. This rebate is deducted from the PNL of the trader, and the function `sendReferrerRebateToStorage` is called to transfer this amount to the storage contract. The function has a check to verify that the amount is not greater than the `totalRewards`, as shown below:

```
function sendReferrerRebateToStorage(uint _amount)
    external override onlyCallbacks {
    require(_amount > 0, "NO_REWARDS_ALLOCATED");
    require(totalRewards >= _amount, "UNDERFLOW_DETECTED");
    IERC20(junior.asset()).safeTransfer(address(storageT), _amount);

    emit ReferralRebateAwarded(_amount);
}
```

As the `_amount` is taken from the trader PNL and not the `totalRewards`, there is no need for the second require statement. The value of `totalRewards` could be less than `_amount`, for example in the case `distributeRewardsWithThreshold` is called, which reduces the value of `totalRewards` as these rewards are distributed. This could lead to `_unregisterTrade` being reverted, causing the closing of trades to be unsuccessful.

Impact

If the closing of trades are reverted, it would in turn lead to failure of stop-loss/take-profit triggers and the liquidations being reverted too.

Recommendations

We recommend removing the require statement.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `61a5c8c2` ↗.

3.2. More rewards allocated to the vault than available

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

Description

When a trade is opened, the `positionSizeUSDC` of the trade is decremented by the fee amount. When a trade is pending, the USDC remains in the storage contract, and it is transferred to the vault manager when the trade is finalized. The `_registerTrade` function calls the `handleDevGovFees` in the storage contract, which transfers the rewards to be allocated to the vault manager.

As the dev fee, referrer rebate, and the governance fees are already in the storage contract, they are not required to be transferred. But the `vaultAllocation - referrerRebate` is transferred to the vault manager. This is the amount that should be increased as the `totalRewards` in the vault manager. However, the function `handleDevGovFees` calls `allocateRewards` with `vaultAllocation` instead of `vaultAllocation - referrerRebate`, as shown below:

```
function handleDevGovFees(
    //...
) external override onlyTrading returns (uint feeAfterRebate) {
    //...
    if (_usdc) IERC20(usdc).safeTransfer(address(vaultManager),
    vaultAllocation - referrerRebate);
    //...
    vaultManager.allocateRewards(vaultAllocation, false);
    //...
}
```

Impact

The rewards allocated to the vault manager are more than the actual rewards transferred.

Recommendations

Call `allocateRewards` with `vaultAllocation - referrerRebate` instead of `vaultAllocation`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `2501eef7` ↗.

3.3. Referrer rebate not paid by traders while closing a position

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

Description

When a trade is closed, fees are deducted from the trader's PNL. The are two different fees: the LP fees and the executor fees. The LP fees are further split into the vault allocation, governance fees, dev fees, and the referrer rebate.

The LP fees also go though a discount process. The discount is applied to the LP fees in the referral contract, which also returns the referrer rebate, as shown below:

```
traderFeesPostDiscount = _fee - (_fee * referralTiers[_tierId].feeDiscountPct)
    / _BASIS_POINTS;
rebateShare = (traderFeesPostDiscount * referralTiers[_tierId].refRebatePct)
    / _BASIS_POINTS;
uint feeDiscount = traderFeesPostDiscount
    * discountTiers[traderTiers[_account]].feeDiscountPct / _BASIS_POINTS;
traderFeesPostDiscount = traderFeesPostDiscount - feeDiscount;
```

Finally, the values `traderFeesPostDiscount` and `rebateShare` are returned in the `_unregister-Trade` function as `feeAfterRebate` and `referrerRebate`. The `feeAfterRebate` actually includes the `referrerRebate` which should be deducted from it to calculate the `vaultAllocation` and the `gov-Fees` but currently its not deducted.

Impact

The referrer rebate is not paid by the trader but the vault manager.

Recommendations

We recommend adding the `rebateShare` to the `totalFees` paid by the trader.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 39bb823b ↗.

3.4. Incorrect type verification in force unregistration

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

Description

The function `forceUnregisterPendingMarketOrder` is used to forcefully unregister a pending market order. It is used as a last resort in case a trader's collateral gets stuck in the storage. It takes the `_id` of the pending market order as an argument and loops through the pending order IDs to find the order with the given `_id`. Then it checks if the `orderType` is `IPriceAggregator.OrderType.MARKET_OPEN`. If it is, it decreases the count of `pendingMarketOpenCount` for that `pairIndex` for the trader; otherwise, it decreases the count of `pendingMarketCloseCount` for that `pairIndex` for the trader.

```
function forceUnregisterPendingMarketOrder(uint _id)
    external override onlyTrading{

//...

for (uint i = 0; i < orderIds.length; i++) {
    if (orderIds[i] == _id) {
        if (orderType == IPriceAggregator.OrderType.MARKET_OPEN) {

    pendingMarketOpenCount[_order.trade.trader][_order.trade.pairIndex]--;
        } else {

    pendingMarketCloseCount[_order.trade.trader][_order.trade.pairIndex]--;
            _openTradesInfo[_order.trade.trader][_order.trade.pairIndex]
            [_order.trade.index]
                .beingMarketClosed = false;
        }
//...
}
```

After the recent changes in the order types, the open market orders are of two types: `MARKET_OPEN` and `MARKET_OPEN_PNL`. The function should check the open orders of both of these types.

Impact

If the order type is `MARKET_OPEN_PNL`, it would decrease the count of `pendingMarketCloseCount` for that `pairIndex` for the trader instead of `pendingMarketOpenCount`.

Recommendations

We recommend checking the order type for both `MARKET_OPEN` and `MARKET_OPEN_PNL` in the function.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `7a5e1dc1` ↗.

3.5. Require check in `distributePnlRewardsFraction` could be improved

Target	VaultManager		
Category	Code Maturity	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The function `distributePnlRewardsFraction` is used to distribute the `pnlRewards`. The VaultManager contract has both the `pnlRewards` and the `totalRewards`; therefore, it is essential to check that the distribution of `pnlRewards` is not depleting the `totalRewards`. The require check in the function only checks that the balance of the contract is greater than the rewards to distribute without taking into account that the balance also includes the `totalRewards` amount.

```
require(IERC20(junior.asset()).balanceOf(address(this)) >
    pnlRewardsToDistribute, "INSUFFICIENT_BALANCE");
```

Impact

If the rewards distributed by the function are greater than the `pnlRewards` (in case the fraction is greater than 100), there would not be enough balance in the VaultManager to cover the `totalRewards` amount.

Recommendations

We recommend modifying the require statement to the following:

```
require(IERC20(junior.asset()).balanceOf(address(this)) - totalRewards >
    pnlRewardsToDistribute, "INSUFFICIENT_BALANCE");
```

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `5a64eddd` ↗.

3.6. Pending `codeOwners` are not deleted in `govSetCodeOwner`

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

Description

The function `govSetCodeOwner` is used to set the code owner of a provided `_code` to a new address. The function deletes the `codes` and the `codeOwners` mapping of the previous code owner. However, it does not delete the `pendingCodeOwners` mapping of `_code` and `codes[_newAccount]`, as shown below:

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

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

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

    delete referrerTiers[account];
    emit GovSetCodeOwner(_code, _newAccount);
}
```

Impact

If the `_code` has a pending code owner, they could claim the ownership of the `_code` after the governance sets the new code owner. This would take away the ownership from the new code owner, which was set by the governance.

Recommendations

We recommend deleting the `pendingCodeOwners` mapping of the `_code` and `codes[_newAccount]` in the `govSetCodeOwner` function.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `67418b0a` ↗.

3.7. Invalid pair index

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

Description

Some functions that use `_pairIndex` as a parameter in PairStorage do not check if a given pair index is valid. You can find many of those patterns in the codebase.

For example, calling the `guaranteedSlEnabled` function with an incorrect pair index will always return true.

```
function guaranteedSlEnabled(uint _pairIndex)
    external view override returns (bool) {
        uint groupIndex = pairs[_pairIndex].groupIndex;
        if(groupIndex == 2 || groupIndex == 3) return false; // Disabled for
    Forex and commodities
        return true;
    }
```

This is because the type of pairs is `mapping(uint256 => Pairs)`, and accessing pairs with an invalid pair index will return the default pair. The default pair will return default values of each member. So in this case, `pairs[_pairIndex].groupIndex` will return zero since the type of `groupIndex` is `uint`. Thus, the return value of a function with an invalid `_pairIndex` will always return true.

Impact

An incorrect pair index may lead to confusing the system.

Recommendations

We recommend checking the index is mapped correctly.

Remediation

This issue has been acknowledged by Avantis Labs, Inc..

3.8. PairInfos could not update the address of PairStorage

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

Description

The Avantis team is planning to migrate the PairStorage contract to a new address. This would require the address of the PairStorage contract to be updated in all the contracts that use it. While there is a `updatePairsStorage` function in the PriceAggregator contract, there is no such function in the PairInfos contract.

Impact

The PairInfos contract is not able to update the address of the PairStorage contract.

Recommendations

We recommend upgrading the PairInfos contract to add a function that would allow to update the `pairsStorage` address in the PairInfos contract.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `951f404c` ↗.

3.9. Initialize functions are front-runnable

Target	All In-scope Contracts		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

All `initialize` functions are front-runnable. The absence of an access-control modifier in these functions means anyone can initialize the contract directly. For example, there is no modifier in the initialize function of PairInfos:

```
function initialize(address _storageT, address _pairsStorage)
    external initializer {
    storageT = ITradingStorage(_storageT);
    pairsStorage = IPairStorage(_pairsStorage);
    liqThreshold =  85;
}
```

Impact

There is no security impact since it will not be used if someone has already initialized this.

Recommendations

We recommend adding an admin role to all `initialize` functions.

Remediation

This issue has been acknowledged by Avantis Labs, Inc..

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 contract inherits PausableUpgradeable, but it is not used

The Referral contract inherits from PausableUpgradeable, but it does not use the pausable functionality. We recommend removing the inheritance if it is not needed.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `40841ace` ↗.

4.2. Broken checks-effects-interactions patterns

We found there are a few instances of broken check-effects-interactions patterns in USDC transfers.

```
function claimFees() external onlyGov {
    IERC20(usdc).safeTransfer(govTreasury, govFeesUSDC);
    IERC20(usdc).safeTransfer(dev, devFeesUSDC);

    emit FeesClaimed(govTreasury, govFeesUSDC, dev, devFeesUSDC);
    devFeesUSDC = 0;
    govFeesUSDC = 0;
}

/**
 * @notice Allows a referrer to claim their rebate.
 */
function claimRebate() external {
    IERC20(usdc).safeTransfer(msg.sender, rebates[msg.sender]);

    emit RebateClaimed(msg.sender, rebates[msg.sender]);
    rebates[msg.sender] = 0;
}
```

These are no security issues at this moment, but could cause an issue in the future if hooks are added to the USDC transfers.

4.3. Ambiguous comments

We can see the `executeLimitOrder` function has an `onlyOperator` modifier, but it is called the "keeper method" in the comments.

```
/**
 * @notice Keeper method to close pending market orders
 * @param orderId The array of orderIDs
 * @param priceUpdateData Pyth price update calldata
 */
function executeMarketOrders(uint[] calldata orderId, bytes[]
    calldata priceUpdateData) external payable onlyOperator{

    for(uint i = 0; i< orderId.length; i++){
        storageT.priceAggregator().fulfill{value:  msg.value}(orderId[i],
    priceUpdateData);
    }
}
```

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `40841ace ↗`.

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: Trading.sol

Function: `closeTradeMarket(uint256 _pairIndex, uint256 _index, uint256 _amount)`

This function 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.

Branches and code coverage

Intended branches

- Generates a new `orderId` for the close-trade market and stores the pending market order.
    - [x] 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 or being closed.
    - [ ] Negative test

- Revert if the leverage of the trade is zero.
    - ☐ Negative test

Function call analysis

- `this.storageT.openTrades(this.__msgSender(), _pairIndex, _index)`
    - **What is controllable?** `__msgSender()`, `_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(this.__msgSender(), _pairIndex, _index)`
    - **What is controllable?** `__msgSender()`, `_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(this.__msgSender())`
    - **What is controllable?** `__msgSender()`.
    - **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(_amount, t.leverage)`
    - **What is controllable?** `_amount` 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; the value is used to compare against the threshold for trade closing.
    - **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.priceAggregator().pairsStorage().getPosType(this.__msgSender(), _pairIndex, _index)`
    - **What is controllable?** `__msgSender()`, `_pairIndex`, and `_index`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Gets the type of this opened position.
    - **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.priceAggregator().getPrice(_pairIndex, OrderType.MARKET_CLOSE_PNL)`
  - **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.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(this.__msgSender( _pairIndex, _index, _amount, 0, 0, False, 0, 0, 0, 0), 0, 0, 0), orderId, False)`
  - **What is controllable?** `__msgSender()`, `_pairIndex`, `_index`, and `_amount`.
  - **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.

Function: `openTrade(ITradingStorage.Trade t, IExecute.OpenLimitOrderType _type, uint256 _slippageP)`

This function 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**: The type of trade to be opened.
- `_slippageP`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The slippage percentage.

Branches and code coverage

Intended branches

- If the order type is `MARKET` or `MARKET_PNL`, store the pending market order.
  - ☑ Test coverage
- If the order type is `LIMIT`, store the open limit order.
  - ☑ Test coverage
- If the take profit and stop loss 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 PairStorage contract, to which calls will be made.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.pendingOrderIdsCount(this.__msgSender())`
  - **What is controllable?** `__msgSender()`.
  - **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.openTradesCount(this.__msgSender(), t.pairIndex)`
    - **What is controllable?** `__msgSender()` 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(this.__msgSender(), t.pairIndex)`
    - **What is controllable?** `__msgSender()` 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(this.__msgSender(), t.pairIndex)`
    - **What is controllable?** `__msgSender()` 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.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.
- `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, False)`
    - **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, False)`
    - **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(this.__msgSender(), address(this.storageT), t.positionSizeUSDC)`
    - **What is controllable?** `__msgSender()` and `t.positionSizeUSDC`.
    - **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(this.__msgSender(), t.pairIndex)`
    - **What is controllable?** `__msgSender()` 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(this.__msgSender(), t.pairIndex, index, t.positionSizeUSDC, t.buy, t.leverage, t.tp, t.sl, t.openPrice, _slippageP, block.number, 0))`
    - **What is controllable?** `__msgSender()`, `t.pairIndex`, `index`, `t.positionSizeUSDC`, `t.buy`, `t.leverage`, `t.tp`, `t.sl`, `t.openPrice`, `block.number`, and `_slippageP`.
    - **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(this.__msgSender(), t.pairIndex, index, _type)`
    - **What is controllable?** `__msgSender()`, `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.
- `aggregator.getPrice(t.pairIndex, OrderType.MARKET_OPEN_PNL)`
    - **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(this.__msgSender(` `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?** `__msgSender()`, `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.

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

This function could be used by traders to update the margin of their open trade.

Inputs

- `_pairIndex`
    - **Control**: Fully controllable.
    - **Constraints**: No constraints.
    - **Impact**: The index of the pair of that trade.
- `_index`
    - **Control**: Fully controllable.
    - **Constraints**: No constraints.
    - **Impact**: The index of the trade.
- `_type`
    - **Control**: Fully controllable.
    - **Constraints**: No constraints.
    - **Impact**: The type of the trade.
- `_amount`
    - **Control**: Fully controllable.
    - **Constraints**: No constraints.
    - **Impact**: The amount to withdraw or deposit.
- `priceUpdateData`
    - **Control**: Fully controllable.
    - **Constraints**: No constraints.
    - **Impact**: The price data for that pair.

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 the `trader` address to the storage contract if it is a deposit call.
  - ☑ 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(this.__msgSender(), _pairIndex, _index)`
  - **What is controllable?** `__msgSender()`, `_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(this.__msgSender(), _pairIndex, _index)`
  - **What is controllable?** `__msgSender()`, `_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.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(this.__msgSender(), _pairIndex, _index, _type, _amount, i.lossProtection, marginFees, t.leverage))`
    - **What is controllable?** `orderId`, `__msgSender()`, `_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.priceAggregator().pairsStorage().getPosType(this.__msgSender(), _pairIndex, _index)`
    - **What is controllable?** `__msgSender()`, `_pairIndex`, and `_index`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Gets the type of this opened position.
    - **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, isPnl)`
    - **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, isPnl)`
    - **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?** N/A.
- `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 _slippageP, uint256 _tp, uint256 _sl)`

This function 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.
- `_slippageP`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Sets the slippage of the limit order.
- `_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 take profit and stop loss 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. (This 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(this.__msgSender(), _pairIndex, _index)`
  - **What is controllable?** `__msgSender()`, `_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.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet, but a former version was deployed. Deployment is targeting Q4 and beyond.

During our assessment on the scoped Avantis contracts, we discovered nine findings. No critical issues were found. Four findings were of high impact, two were of medium impact, two were of low impact, and the remaining finding was informational in nature.

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.