TRADING ACCOUNT BRANCH CONTRACT REVIEW

INTRODUCTION:

Zaros is a perpetual decentralised exchange that gives users on its platform the opportunity to speculate on the differences in price of assets available on the Zaros platform, without users necessarily having to own it.

The Trading Account Branch contract is a component of the Zaros perpetual decentralised exchange and it is aimed at providing some form of management to users participation in the Zeros ecosystem through features such as trading account management, margin collateral operations, referral system setup, leverage calculations and so many more.

SETUP:

SPDX-License-Identifier: UNLICENSED
This is required in Solidity files to indicate the license type under which the code is shared.

pragma solidity 0.8.25: This is the setup that defines the version of language that the solidity compiler uses to compile the contract.

DEPENDENCIES:

For this contract to work as efficiently as possible, it relied on a few imports. These imports are discussed below in order to emphasise their importance and highlight their contribution to the Trading Account Branch:

IAccountNFT.sol: The IAccountNFT interface acts as a gateway for interacting with the AccountNFT contract available within the account-nft directory of the *Zaros* repository. To interact with the IAccountNFT, the mint function signature was declared in the IAccountNFT, where the mint function intends to take in an address as argument, and a tokenId. This IAccountNFT will later be used to get reference to the *NFT* account token within the Trading Account Branch contract, and later allow a user to mint NFT.

import { IAccountNFT } from "@zaros/account-nft/interfaces/IAccountNFT.sol";

Errors.sol: The Errors.sol dependency imported into the Trading Account Branch contract from the utils directory within the zeros project. The Errors.sol dependency serves as a library that defines custom errors which are used to notify the user of the platform about anomalies or issues that must have occurred in the course of their activity on the platform. The main aim for this is to have reusable and defined error messages across all facets of the Zaros platform. Meanwhile, for the Trading Account Branch, this library was used to make error notifications in areas where the provided referral code is invalid, when an input value is Zero, when an inputted amount value exceeds the 18 decimals, and when user are trying to deposit a collateral type that isn't specified within the liquidation priority configuration.

import { TradingAccount } from "@zaros/perpetuals/leaves/TradingAccount.sol";

TradingAccount.sol: This is a library written within the leaves directory of the zaros project, this library is intended to provide a form of risk management for margin trading that takes place on the Zaros project, actually because it's a library, other perpetual futures market can as well use it. The TradingAccount library basically helps each trading account to maintain adequate collateral, while adhering to the position limits, it also properly calculates the margin requirements based on collateral configurations, which inturn enhances the stability and safety of the Zaros platform for its users. For the TradingAccountBranch, this dependency was used to check if the Trading Account exists, also used to get the margin collateral balance of the collateral type passed in, while also used to get the total margin collateral value in USD.

GlobalConfiguration.sol: This dependency serves as a library that helps in the management of the global configuration settings needed for use within the Trading Account Branch. For our Trading Account Branch, it leveraged this library to store global settings in one location, and help in setting collateral liquidation principles.

import { GlobalConfiguration } from "@zaros/perpetuals/leaves/GlobalConfiguration.sol";

PerpMarket.sol: This dependency serves as a library that helps the zaros decentralised exchange to offer the possibility or features

for users to open, close, long and short positions within zaros dex. For our Trading Account Branch the PerpMarket offers the opportunity to ensure dynamic pricing, funding rates and offer fees.

Position.sol: This dependency offers the Trading Account Branch contract a library which is used to manage a number of operations within the zaros decentralised exchange, it does this by using advanced storage technique and mathematical calculations to manage the state and properties of trading positions. For our Trading Account Branch, this dependency was used to track margin calculations, profit/loss and adjust funding.

MarginCollateralConfiguration.sol: Since we have margin collateral used within the Zaros Decentralised exchange, it is necessary that we have a mechanism in place to manage that. Hence, the Margin Collateral Configurations is a library that is used to manage configurations for margin collateral.

CustomReferralConfiguration.sol: This dependency is used to provide a means of managing custom referral configurations, it does this by storing and retrieving information about the referral from storage location.

Referral.sol: This particular library handles both regular and custom referral codes for each account, and also helps in managing referral codes.= within the Trading Account Branch.

EnumerableSet: This dependency serves as a library from OpenZeppelin, it helps in the provision of a data structure that is in turn used for managing collections, or sets of elements that are more than one, or elements that can be looped over. With this dependency we can carry out operations such as addition, removal of those elements. A major use case for this within the Trading Account Branch was to track active markets through tradingAccount.activeMarketsIds. This basically ensures that no duplicate entry exists.

IERC20.sol: This dependency serves as an interface for the ERC20 token, in it we have functions signatures that have already been predefined as standards, functions signature such as transfer,

approve and transferFrom are defined in it. For our Trading Account Branch, the IERC20 interface was used to handle activities involving token operation, such as token transfer, deposits and withdrawals.

SafeCast: This dependency provides functions that are used for safely converting between different integer types such as uint256 to uint128, this is done to prevent overflow errors. For the Trading Account Branch contract, we need this to ensure no duplicate entries.

SafeErc20: This dependency is a library that helps to provide some level of safety around ERC20 token. It does this by wrapping around ERC20 token operations, adding checks to prevent errors, and so that they do not fail unexpectedly.

PRB MATHS DEPENDENCIES:
This dependencies that will be listed below helps to provide high precision fixed point arithmetic, thereby allowing for decimal calculations

UD60x18 and SD59x18:

These types represent unsigned and signed decimal values with 18 decimal places, crucial for precise financial calculations. UD60x18 is used for margin values (e.g., marginCollateralBalanceX18), while SD59x18 is used for potentially negative values, such as unrealized profits and losses.

ZERO as UD60x18_ZERO and ZERO as SD59x18_ZERO:

These constants are used to initialise zero values for UD60x18 and SD59x18 types to avoid underflows or overflows in financial calculations.

Unary:

The unary function allows type conversion and manipulation, such as calculating position sizes and updating notional values using signed decimal representations.

CODE STRUCTURE REVIEW AND DEFINITION

contract TradingAccountBranch{}: this is declaring a contract name TradingAccountBranch, which can as well be defined as a blueprint, just the same way we have class in other object oriented programming, this also serves as defining a blueprint and with this we can have multiple and many instance of this contract created, in the curly braces, we can have the contract data and logic defined within.

using EnumerableSet for *: This line defined within the TradingAccountBranch contract uses the imported EnumerableSet library from Openzeppelin contracts functions and applies them automatically on all forms of data types. So this means functions within the EnumerableSet (functions like adding, removing, etc) can be applied on other data types within the contract even though those functions are not defined as functions within the data type such as address, bytes32, uint256.

using TradingAccount for TradingAccount.Data: This line helps to extend the functions defined in the TradingAccount library on the TradingAccount.Data, and we can use it on it as if they were its own.

using PerpMarket for PerpMarket.Data: This line helps in inheriting the PerpMarket library function on the PerpMarket.Data, the PerpMarket.Data is a struct defined within the PerpMarket library, and using this prevents us from having functions littered around, hence it makes our code cleaner and more intuitive.

using Position for Position.Data: This does the same thing as mentioned above, the Position.Data Struct is able to use functions defined in the Position library.

using SafeCast for uint256: This lines sets up the contract uint256 data type to associate with SafeCast library, this basically does the job of casting integers of type uint256 to smaller types of either uint8 or uint16 without having issues related to overflow. This is very much needed to avoid any form of integer overflow issues when carrying out operations.

using SafeERC20 for IERC20: With this line, it means we are calling the SafeERC20 library on IERC20 tokens and hence they can use functions within the SafeERC20 library.

using GlobalConfiguration for GlobalConfiguration.Data: This line has GlobalConfiguration.Data Struct inherits the functions from GlobalConfiguration. Since the GlobalConfiguration.Data holds configurations that apply globally within our contract, with it inheriting from GlobalConfiguration, this means we can efficiently manage and access these configurations across the contract.

using MarginCollateralConfiguration forMarginCollateralConfiguration .Data: Same as the previous lines, this line also allows MarginCollateralConfiguration.Data to use the functions available within the MarginCollateral Configuration library.

using Referral for Referral.Data: This line attaches the Referral library functions to Referral.Data struct containing relevant data such as referral bonuses and or referred users.

EVENTS USED WITHIN THE TRADING CONTRACT

LogCreateTradingAccount(uint128 tradingAccountId, address sender): this line presents an event that is emitted upon the successfully creation of new trading account, it emits data such as the trading account Id, and the address of the creator of the account.

LogDepositMargin(address indexed sender, uint128 indexed tradingAccountId, address indexed collateralType, uint256 amount): This line contains an events that is emitted when a creator of trading account deposits amount of collateral type into trading account id, the emitted data includes:
the address of the creator (who is the caller), the account Id of the trading Account created, the address of the margin collateral, and the amount of margin collateral deposited.

LogWithdrawMargin(address indexed sender, uint128 indexed tradingAccountId, address indexed collateralType, uint256 amount): This line emits upon successfully withdrawal of collateral type from the trading account, and once it is successful it emits the address of the user making the withdrawal, the Id of the trading account, the margin collateral address, the token amount of margin collateral withdrawn.

LogReferralSet(address indexed user, address indexed referrer, bytes referralCode, bool isCustomReferralCode): This line emits when a referral code is set Successfully, and upon this it emits the

following data, the address of the user being referred, the address of the user referring, the referring code, and a boolean value to confirm if the referral code is of custom type

FUNCTIONS WITHIN THE TRADING CONTRACT

getTradingAccountToken: This function gets the contract address of the trading account NFTs, and it does this by defining a public function that reads from the Global configuration. The global configuration is basically a library that contains the global configuration settings and with the load function called on it, it retrieves the current configurations data from storage into the memory for access, in this case it retrieves the address of trading AccountToken.

getAccountMarginCollateralBalance: This is a function within the Trading Account Branch, and it takes in two parameters, a tradingAccountId, and an address of a collateral type. The function is expected to read and return the balance of an address passed as collateral type for a specific trading account, this collateral type passed is an ERC20 address. The function carried out this by calling on the loadExisting function defined within the TradingAccount library, the loadExisting function takes in the trading accountId passed and it retrieves data about the trading account stored, upon doing this the getMarginCollaterealBalance function was called on the instance of the trading account that was created so as to get the margin collateral balance of the collateral type that was passed, and then it was returned.

getAccountEquityUsd: This function retrieves the account's total equity in USD. It takes a tradingAccountId as input and performs a number of actions, such as: Loading the existing trading account data using the *TradingAccount.loadExisting* function, retrieving the account's unrealized profit or loss in USD using the getAccountUnrealizedPnlUsd function from the loaded trading account data, and then calls the getEquityUsd function (also from the loaded trading account data) to calculate the total equity by considering the unrealized profit and loss, at the end of the function running successfully, it returns the calculated equity value in USD.

getAccountMarginBreakdown: This function provides a detailed breakdown of the account's margin status. It takes a tradingAccountId as input and returns a number of values, such as themarginBalanceUsdX18,initialMarginUsdX18, maintenanceMarginUsdX18, availableMarginUsdX18, and they are all in 18 decimals places as precision. The marginBalanceUsdX18 serves as the total account margin balance in USD, while the initialMarginUsdX18 value returned serves as the total initial margin required for open positions in USD, while the maintenanceMarginUsdX18, serves as the total maintenance margin required for open positions in USD, this is as well referred to as the minimum margin needed to avoid liquidation, while the fourth return value serves as the withdrawable margin balance in USD, and it is calculated by subtracting initial margin from total margin. While explaining the function logic, it is important to note that the first line of the logic loads up the existing trading account data, then it retrieves the unrealized Profit and Loss in USD, then it calls getMarginBalanceUsd to calculate the total margin balance considering the Profit and Loss, and after doing this it loops through all active markets associated with the trading account, then it loads the market data and the corresponding position data, after which it retrieves the index price and calculates the mark price for the position, then it calculates the notional value of the position (position size multiplied by mark price), It then uses the getMarginRequirement function (from the Position library) to determine the initial and maintenance margin requirements for the specific position based on its notional value and the market's configuration, It then accumulates the initial and maintenance margin requirements across all active markets, while finally, it calculates the available margin by subtracting the initial margin from the total margin balance.

getAccountTotalUnrealizedPnL: This external function basically retrieves the total unrealized profit or loss across all open positions in the trading account. It simply loads the existing trading account data and calls the getAccountUnrealizedPnlUsd function to obtain the unrealized PnL value in USD.

getAccountLeverage: This function is an external function that is aimed at calculating and returning the ratio of the leverage for a specific trading account, which is identified based on the trading account Id. It does this by first defining an external function and

ensuring that it is accessible from publicly but not internally, it accepts an id, which uniquely identifies the trading account and then returns a value of type UD60x18. The next line had us load the data associated with the specified trading account from storage using the loadExisting function in the TradingAccount library. Once this is done, the tradingAccount variable now points to the data, while allowing access to relevant properties within it. By using storage, the function directly references the on-chain data instead of creating a copy in memory, an efficient choice especially if TradingAccount.Data is a larger or more complex structure. This makes tradingAccount a reference to the account's stored data, saving on computational costs and providing a direct link to the live data on the blockchain.

Next, the function calculates the unrealized profit and loss (PnL) across all open, or active, positions within the account. To achieve this, it calls the method getAccountUnrealizedPnlUsd() on tradingAccount, which returns the unrealized PnL in USD and stores it in activePositionsUnrealizedPnlUsdX18. This variable captures the potential gains or losses from the account's active positions, which are yet to be realized since these positions are still open and haven't been closed or settled.

Finally, the function uses this unrealized PnL value to compute the total equity of the account by calling getEquityUsd on tradingAccount, passing activePositionsUnrealizedPnlUsdX18 as a parameter. This call combines the account's current balance and its unrealized PnL to calculate its total equity. The function then returns this result, providing the full equity in USD as the function's output. This equity value represents the total financial standing of the trading account, factoring in both its current assets and the potential gains or losses from its open positions.

getPositionState: This function is intended to retrieve the current state of a trading position within a specified market for a given trading account. It takes three parameters as input: `tradingAccountId`, which identifies the trading account in question; `marketId`, which specifies the market where the position is held; and `indexPrice`, representing the latest index price for the market. This function is marked as `external`, allowing it to be called from outside the contract, and `view`, indicating that it only reads data without modifying any on-chain state. The function

returns a `Position.State` structure in memory, representing the current status of the position, including factors such as margin requirements, market prices, and any applicable fees.

To begin, the function loads the market data for the specified `marketId`. This is achieved by creating a `storage` reference variable named `perpMarket`, which points to `PerpMarket.Data`. The function calls `PerpMarket.load(marketId)` to fetch and reference the on-chain market data directly. By using `storage`, the function accesses the data directly on-chain, rather than creating a copy in memory, making it more efficient and ensuring it operates on the latest data.

The function then loads the trading position data for the specified account and market by creating another `storage` variable named `position`, which references `Position.Data`. It calls `Position.load(tradingAccountId, marketId)` to fetch the position data associated with that specific account and market. Like the previous `storage` variable, this direct reference is efficient and necessary for handling potentially complex structures directly.

Next, the function calculates the `markPriceX18`, the real-time market price for the position, using the `perpMarket.getMarkPrice()` method. This call takes two parameters: the position size (converted to a signed decimal format) and the `indexPrice` (converted to a fixed-point unsigned decimal format). The resulting `markPriceX18` value is a crucial component, as it represents the actual price of the asset in this particular market at the current time.

Following this, the function calculates the ‘fundingFeePerUnit`, which represents the fee per unit of the position due to funding rates. This is determined by calling `perpMarket.getNextFundingFeePerUnit()` with two arguments:

the current funding rate retrieved from `perpMarket.getCurrentFundingRate()` and the `markPriceX18` value calculated earlier. The `fundingFeePerUnit` accounts for any funding fees that will affect the position over time.

Finally, the function populates and returns the `positionState`, which encapsulates the entire state of the position at this point. It does so by calling `position.getState()`, passing in several parameters: the initial and maintenance margin rates from

`perpMarket.configuration` (both converted to fixed-point unsigned decimal format), the `markPriceX18`, and the `fundingFeePerUnit`. This final state includes all necessary information about the position, like margin requirements, the latest market price, and funding fees, providing a comprehensive snapshot of the position’s status within the specified market. The function then returns this `positionState` structure as its output.

createTradingAccount: this function is responsible for creating a new trading account for a user. When called, it first accesses a storage slot dedicated to the global configuration settings, where it increments the next available account ID and assigns it to this new account. It then sets up the account by interacting with an Account NFT contract, represented by the IAccountNFT interface. This contract allows the system to mint a unique token that identifies the account, linking it directly to the owner's address. Following this setup, an event logs the creation of the trading account, recording the account ID and the owner's address.

Referral codes, which might be used by the account holder, play an important role in this function. If a user provides a referral code when creating their trading account, the function checks if the user has already associated a referral code with their address. If they haven't, the system examines the provided referral code to determine if it's a custom code. For custom codes, it verifies that the code is valid by checking for an existing referrer in the custom referral configuration. If valid, the code is associated with the user's account. Otherwise, the function decodes the referral code as an address and ensures the user is not referring to themselves. If everything checks out, an event logs this referral setup, capturing details about the referral and its customization status.

createTradingAccountAndMulticall: This function expands on the trading account creation process, enabling the account holder to make multiple contract calls immediately after their account is created. This feature is useful for users who want to set up various parameters or initiate trades right after account creation. The function first creates the trading account by calling createTradingAccount and storing its ID. With the account ID established, the function iterates over each call data payload in data, a collection of calldata to be used in further contract calls. Each payload is modified to include the new account ID, and the

function calls these modified payloads using delegatecall. If any call fails, the function immediately reverts, preserving the atomicity of this process. If successful, the function returns an array containing the results of each call.

depositMargin: With this function a user can deposit margin collateral into a specified trading account. The function begins by loading the margin collateral configuration data for the specified collateral type. To ensure the account exists, it retrieves the relevant trading account data without enforcing the account owner's identity, allowing anyone to deposit collateral on behalf of the account holder if they wish. The function performs a series of checks, including converting the deposit amount into a standardised format with 18 decimals, verifying the deposit cap, and ensuring that a liquidation priority has been set for this collateral. After these checks, the collateral tokens are transferred from the user to the contract using the safeTransferFrom method, ensuring safe token handling. The deposit is recorded in the trading account, and an event logs the margin deposit details, capturing the depositor’s address, account ID, collateral type, and amount.

withdrawMargin: This function is designed for account owners to withdraw available margin collateral from their trading accounts. Starting with the margin collateral configuration, it loads the necessary data for the specified collateral type. The function also retrieves the trading account information, enforcing that only the account owner can initiate the withdrawal. Similar to the deposit function, the withdrawal amount is converted to an 18-decimal format and checked to ensure the user has enough collateral to cover the withdrawal. Additionally, the function calculates the account’s total margin requirements and unrealized profit or loss to verify that the withdrawal will not reduce the margin balance below the initial requirement. This check prevents immediate liquidation due to insufficient margin, enhancing account safety. Upon validation, the specified amount of collateral is transferred back to the user, and a withdrawal event logs the action.

notifyAccountTransfer: This function enables the transfer of account ownership when an NFT associated with the trading account changes hands. This transfer function can only be called by the Account NFT contract, ensuring controlled ownership updates. When called, it updates the owner’s address within the trading account’s data,

recording the new owner's address and linking it to the respective account ID.

getUserReferralData: This function provides a straightforward way to access a user's referral code and determine if it's a custom referral. By fetching the referral data from the storage, it returns both the referral code and a boolean indicating the customization status, allowing users or external functions to query referral details easily.

_requireAmountZero: This function serves as a utility to enforce that an amount is not zero. If the amount passed is zero, the function reverts with a specific error, preserving data integrity and ensuring non-zero values are processed.

_requireEnoughDepositCap: This function checks that the amount to be deposited does not exceed the collateral deposit cap. It sums the new amount with the total deposited amount and ensures it remains below the cap. If the cap is exceeded, the function reverts, preventing excess deposits and maintaining deposit limits.

_requireCollateralLiquidationPriorityDefined: This function checks if the specified collateral type is listed in the global configuration's liquidation priority list. If not, the function reverts with an error, ensuring only approved collateral types are used.

_requireEnoughMarginCollateral: This function validates that the user has enough collateral of the specified type. By checking the collateral balance against the requested amount, it confirms that only deposited collateral is available for withdrawal. If insufficient collateral exists, the function reverts, ensuring proper margin requirements.

_onlyTradingAccountToken: This function restricts certain actions to the Account NFT contract, enforcing that only the designated contract can execute account-related functions. If an unauthorised address attempts to call, the function reverts, maintaining secure access control within the contract.