ZAROS
Security Review

JANUARY, 2025

www.chaindefenders.xyz
https://x.com/ChDefendersEth

Lead Auditors

PeterSR

0x539.eth

Table of Contents

Protocol Summary

Zaros is a Perpetuals DEX powered by Boosted (Re)Staking Vaults. It seeks to maximize LPs yield generation, while offering a top-notch trading experience on Arbitrum (and Monad in the future).

Disclaimer

The ChainDefenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of 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.

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

Scope

Id	Files in scope
1	DebtSettlementKeeper.sol
2	FeeConversionKeeper.sol
3	UsdTokenSwapKeeper.sol
4	CreditDelegationBranch.sol
5	FeeDistributionBranch.sol
6	MarketMakingEngineConfigurationBranch.sol
7	StabilityBranch.sol
8	VaultRouterBranch.sol
9	IEngine.sol
10	AssetSwapPath.sol
11	Collateral.sol
12	CreditDelegation.sol
13	DexSwapStrategy.sol
14	Distribution.sol
15	LiveMarkets.sol
16	Market.sol
17	MarketMakingEngineConfiguration.sol
18	StabilityConfiguration.sol
19	Swap.sol
20	UsdTokenSwapConfig.sol
21	Vault.sol
22	WithdrawalRequest.sol
23	MarketMakingEngine.sol
24	LiquidationBranch::liquidateAccounts
25	SettlementBranch::_fillOrder

Id	Files in scope
26	IReferral.sol
27	CustomReferralConfiguration.sol
28	ReferralConfiguration.sol
29	Referral.sol
30	BaseAdapter.sol
31	CurveAdapter.sol
32	UniswapV2Adapter.sol
33	UniswapV3Adapter.sol
34	ICurveSwapRouter.sol
35	IDexAdapter.sol
36	ISwapAssetConfig.sol
37	IUniswapV2Router01.sol
38	IUniswapV2Router02.sol
39	IUniswapV3RouterInterface.sol
40	IUniswapV3SwapCallback.sol
41	EngineAccessControl.sol
42	Whitelist.sol
43	ZlpVault.sol

Roles

Id	Roles
1	Market Making Engine
2	System Keeper
3	Chainlink Automation Forwarder
4	USDz swapper
5	LPs
6	Clients

Executive Summary

Issues found

Severity	Count	Description
High	5	Critical vulnerabilities
Medium	8	Significant risks
Low	2	Minor issues with low impact
Informational	0	Best practices or suggestions
Gas	0	Optimization opportunities

Findings

High

[HIGH-01] getTotalCreditCapacityUsd Does Not Support Low Decimals

Summary

In `Vault` there is a function `getTotalCreditCapacityUsd` used to fetch the vault's total credit capacity allocated to the connected markets. However, currently it doesn't support low decimal tokens.

Vulnerability Details

In `Vault` there is a function `getTotalCreditCapacityUsd` used to fetch the vault's total credit capacity allocated to the connected markets. However, currently it doesn't support low decimal tokens.

```
function getTotalCreditCapacityUsd(Data storage self) internal view
    returns (SD59x18 creditCapacityUsdX18) {
    // load the collateral configuration storage pointer
    Collateral.Data storage collateral = self.collateral;

    // fetch the zlp vault's total assets amount
    UD60×18 totalAssetsX18 = ud60×18(IERC4626(self.indexToken).
   totalAssets());

    // calculate the total assets value in usd terms
    UD60×18 totalAssetsUsdX18 = collateral.getAdjustedPrice().mul(
   totalAssetsX18);

    // calculate the vault's credit capacity in usd terms
    creditCapacityUsdX18 = totalAssetsUsdX18.intoSD59×18().sub(
   getTotalDebt(self));
}
```

Let's take a look at this line:

```
UD60×18 totalAssetsX18 = ud60×18(IERC4626(self.indexToken).totalAssets
    ());
```

If our `indexToken` has decimals less than 18 then the whole calculation will be wrong.

Let's have the following scenario:

1. `indexToken` has 6 decimals.
2. The total assets are 10 tokens.
3. `totalAssetsX18` will be 10e6.
4. The function will return the value in 6 decimals and not the desired 18.

Impact

Wrong credit capacity calculation will lead to wrong checks performed in `VaultRouterBranch::redeem` which in itself will lead to the check for unlocked credit capacity not working.

Tools Used

Manual Review

Recommendations

Convert the `IERC4626(vault.indexToken).totalAssets()` first to an 18 decimal value before performing the calculation to ensure that all `indexToken`s with decimals less than or equal to 18 will work correctly.

[HIGH-02] `recalculateVaultsCreditCapacity` Will Revert When Vault Has Lower `creditCapacity` Than The Previous One

Summary

During the `recalculateVaultsCreditCapacity` in one of the underlying function `_updateCreditDelegations`, this function will iterate through all of the markets for the current vault. If the `previousCreditDelegationUsdX18` was higher than the current one, logic will revert.

Vulnerability Details

Let's have the following situation:

1. We have 1 Market and this market has only vault, also the vault is connected just to this one.
2. Let's have for the `previousCreditDelegationUsdX18` = 80 and for the current `newCreditDelegationUsdX18` = 70, because for example user withdrawn some of the funds or there is the drop of the price of the asset.
3. The above values will revert due to 70 - 80 = -10, but the value holders are using UD60x18, which is the unassigned value.

Impact

The current implementation of the _updateCreditDelegations function can lead to unexpected reverts when the `newCreditDelegationUsdX18` is less than the `previousCreditDelegationUsdX18`. This occurs because the function uses UD60x18 (unsigned decimal) for calculations, which cannot handle negative values. This will make the vault and respectively the market unusable, until the `newCreditDelegationUsdX18` is less than the `previousCreditDelegationUsdX18`.

Tools Used

Manual Review

Recommendations

Consider changing the logic to use SD59x18:

```
SD60x18 newCreditDelegationUsdX18 = vaultCreditCapacityUsdX18.gt(SD59x18_ZERO)
-                     ? vaultCreditCapacityUsdX18.intoUD60x18().mul(creditDelegationShareX18)
+                    ? vaultCreditCapacityUsdX18.intoSD59x18().mul(creditDelegationShareX18)
                     : SD59x18_ZERO;

                // calculate the delta applied to the market's total delegated credit
-                 UD60x18 creditDeltaUsdX18 = newCreditDelegationUsdX18.sub(previousCreditDelegationUsdX18);
+                 SD60x18 creditDeltaUsdX18 = newCreditDelegationUsdX18.sub(previousCreditDelegationUsdX18);
```

[HIGH-03] Missing Multiplication In The Logic

Summary

During the execution of `getVaultAccumulatedValues`, the protocol calculates the updated vault shares for a given market. However, if the market has multiple vaults, the calculation of WETH rewards is incorrect.

Vulnerability Details

Scenario

1. Assume there are two vaults, both delegated with an equal amount of 50.
2. The market distributes 2 WETH as rewards.
3. The current implementation incorrectly assigns 2 WETH to each vault, resulting in a total of 4 WETH, which exceeds the available rewards.

Root Cause

- The existing calculation does not consider the vault's proportional share of the total market delegation.
- As a result, each vault receives the full WETH reward, rather than a fraction based on its share.

Impact

- Over-distribution of WETH rewards, leading to an incorrect reward allocation.
- Potential financial inconsistencies, as the protocol attempts to distribute more rewards than available.
- Vulnerability to misallocation exploits, affecting fair distribution among vaults.

Tools Used

Manual Review

Recommendations

Modify the reward calculation to ensure proportional distribution among multiple vaults:

```
   usdcCreditChangeX18 = !lastVaultDistributedUsdcCreditPerShareX18.isZero()
             ? ud60x18(self.usdcCreditPerVaultShare).sub(lastVaultDistributedUsdcCreditPerShareX18).mul(
                  vaultCreditShareX18
             )
             : UD60x18_ZERO;

-   wethRewardChangeX18 = ud60x18(self.wethRewardPerVaultShare).sub(lastVaultDistributedWethRewardPerShareX18);
+.   wethRewardChangeX18 = ud60x18(self.wethRewardPerVaultShare)
+        .sub(lastVaultDistributedWethRewardPerShareX18)
+        .mul(vaultCreditShareX18);
```

[HIGH-04] Locked Capacity Check Not Working

Summary

The `VaultRouterBranch`::`redeem` function is used to redeem a given amount of index tokens in exchange for collateral assets from the provided vault, after the withdrawal delay period has elapsed. In the end it has a check trying to ensure that there is enough unlocked credit capacity of the vault so it doesn't leave it insolvent. However, that check is broken.

Vulnerability Details

The `VaultRouterBranch`::`redeem` function is used to redeem a given amount of index tokens in exchange for collateral assets from the provided vault, after the withdrawal delay period has elapsed. In the end it has a check trying to ensure that there is enough unlocked credit capacity of the vault so it doesn't leave it insolvent. However, that check is broken.

```
if (
ctx.creditCapacityBeforeRedeemUsdX18.sub(vault.getTotalCreditCapacityUsd()).lte(
        ctx.lockedCreditCapacityBeforeRedeemUsdX18.intoSD59×18()
    )
) {
    revert Errors.NotEnoughUnlockedCreditCapacity();
}
```

Let's have the following situation:

1. `creditCapacityBeforeRedeemUsdX18` is 50000
2. `vault.getTotalCreditCapacityUsd()` is 10000
3. `lockedCreditCapacityBeforeRedeemUsdX18` is 20000
4. `creditCapacityBeforeRedeemUsdX18 - vault.getTotalCreditCapacityUsd()` is 40000

   This means that 10000 are still left which is less than the locked capacity of 20000
5. However, the check is not working correctly and the transaction will not revert

Impact

This check is used to ensure that the vault can continue operating with its markets and not be left insolvent. With it being broken the vault can become insolvent or not have enough locked capacity needed to operate with its connected markets.

Tools Used

Manual Review

Recommendations

Change the check to this one:

```
if (
    vault.getTotalCreditCapacityUsd().lt(
        ctx.lockedCreditCapacityBeforeRedeemUsdX18
    )
) {
    revert Errors.NotEnoughUnlockedCreditCapacity();
}
```

[HIGH-05] If User Stakes Again, He Will Lose All Of His Rewards

Summary

Currently in `VaultRouterBranch` users can stake their shares to receive rewards in terms of WETH. If user has some of his shares staked and he decide to stake some more, but there is a reward accumulated, he will lose the reward if he stakes again, before claiming the reward.

Vulnerability Details

Let's have the following scenario:

1. User A has 100 shares and he decides to stake 50 shares.

2. After some period, some rewards are accumulated and the user is eligible to claim it.
3. But he decides to stake the other 50, which will lead in situation where `accumulateActor` will update the state of the actor and accumulated rewards will be lost.
4. User will be not able to claim the rewards for the first 50 tokens, which he staked.

The problem lies that the `accumulateActor` function will call underlying this function:

```
function _updateLastValuePerShare(
        Data storage self,
        Actor storage actor,
        UD60x18 newActorShares
    )
        private
        returns (SD59x18 valueChange)
    {
        valueChange = _getActorValueChange(self, actor);

        actor.lastValuePerShare = newActorShares.eq(UD60x18_ZERO) ? int256(0) : self.valuePerShare;
    }
```

The function will set `lastValuePerShare` to the current one and the user will be not able to claim anymore `valueChange`.

Impact

User will be not able to claim their rewards and this will lead to lost of reward.

Tools Used

Manual Review

Recommendations

Consider implementing similar mechanism to the one in unstake:

```
UD60x18 amountToClaimX18 = vault.wethRewardDistribution.getActorValueChange(actorId).intoUD60x18();

// reverts if the claimable amount is NOT 0
if (!amountToClaimX18.isZero()) revert Errors.UserHasPendingRewards(actorId, amountToClaimX18.intoUint256());
```

Medium

[MEDIUM-01] `checkFeeDistributionNeeded` Will Not Work With Low Decimal Assets

Summary

The `FeeConversionKeeper` is a keeper used for automatic fee distribution. It works with two main functions - `checkUpkeep` and `performUpkeep`. When `checkUpkeep` returns true, `performUpkeep` executes. Inside `checkUpkeep` there is a call to `checkFeeDistributionNeeded` to decide whether a distribution is needed or not. However, the function does not work as intended because it makes a call to `FeeDistributionBranch`::`getAssetValue` and expects the value to be in base 18. The function returns "the calculated value of the asset in its native units".

Vulnerability Details

The `FeeConversionKeeper` is a keeper used for automatic fee distribution. It works with two main functions - `checkUpkeep` and `performUpkeep`. When `checkUpkeep` returns true, `performUpkeep` executes. Inside `checkUpkeep` there is a call to `checkFeeDistributionNeeded` to decide whether a distribution is needed or not. However, the function does not work as intended because it makes a call to `FeeDistributionBranch`::`getAssetValue` and expects the value to be in base 18. The function returns "the calculated value of the asset in its native units".

```
/// @notice Calculates the value of a specified asset in terms of its collateral.
/// @dev Uses the asset's price and amount to compute its value.
/// @param asset The address of the asset.
/// @param amount The amount of the asset to calculate the value for.
/// @return value The calculated value of the asset in its native units.
```

```
function getAssetValue(address asset, uint256 amount) public view returns (uint256 value) {
    // load collateral
    Collateral.Data storage collateral = Collateral.load(asset);

    // get asset price in 18 dec
    UD60x18 priceX18 = collateral.getPrice();

    // convert token amount to 18 dec
    UD60x18 amountX18 = collateral.convertTokenAmountToUd60x18(amount);

    // calculate token value based on price
    UD60x18 valueX18 = priceX18.mul(amountX18);

    // ud60x18 -> uint256
    value = collateral.convertUd60x18ToTokenAmount(valueX18);
}
```

And

```
/// @notice Checks if fee distribution is needed based on the asset and the collected fee amount.
/// @param asset The address of the asset being evaluated.
/// @param collectedFee The amount of fee collected for the asset.
/// @return distributionNeeded A boolean indicating whether fee distribution is required.
function checkFeeDistributionNeeded(
    address asset,
    uint256 collectedFee
)
    public
    view
    returns (bool distributionNeeded)
{
    // load keeper data from storage
    FeeConversionKeeperStorage storage self = _getFeeConversionKeeperStorage();

    /// get asset value in USD
    uint256 assetValue = self.marketMakingEngine.getAssetValue(asset, collectedFee);

    // if asset value GT min distribution value return true
    distributionNeeded = assetValue > self.minFeeDistributionValueUsd;
}
```

As we can see the `minFeeDistributionValueUsd` is a fixed value. However, different assets have different collaterals which have different decimals. This means that the current logic will work only for assets with collaterals with decimals equal to 18. If they are less than the `assetValue` will never be enough to trigger a distribution.

Impact

Non working `FeeConversionKeeper` which means non converted fees to WETH in the cases in which the decimals of the collateral of a given asset is less than 18.

Tools Used

Manual Review

Recommendations

Either make a call to `collateral.convertTokenAmountToUd60×18` for the returned value of `getAssetValue` or add a new function (for example `getAssetValueX18`) that will do this internally and return the value in base 18.

[MEDIUM-02] Wrong Values Of Newly Added `vault`

Summary

When a new vault is added for a market, a `CreditDelegation` struct is created for each tuple of Market and Vault. Initially, all values in the `CreditDelegation` struct are set to `0`. This leads to a significant issue because, currently, when the values of `CreditDelegation` are `0`, they can only be updated if there is some `WETH` available for distribution. If no `WETH` is available, the debt for the vault will not be distributed until any `WETH` becomes available. On the other hand if there is a WETH available at the beginning, the protocol will try to distrbute share of it to the vault, even if it joins now. This happens because the calculation involves subtracting `wethRewardPerVaultShare - lastVaultDistributedWethRewardPerShareX18`, and `lastVaultDistributedWethRewardPerShareX18` is initially `0`.

Vulnerability Details

The issue arises in the debt distribution mechanism for new vaults. When a new vault is added, the `CreditDelegation` struct is initialized with default values of `0`. The debt distribution logic relies on the difference between `wethRewardPerVaultShare` and `lastVaultDistributedWethRewardPerShareX18`. Since `lastVaultDistributedWethRewardPerShareX18` is `0` initially, the system will attempt to distribute all WETH accumulated since the market's inception. This can lead to incorrect debt calculations and unfair distribution of rewards.

Key Points:

1. Initialization Issue: New vaults start with `lastVaultDistributedWethRewardPerShareX18 = 0`.
2. Debt Distribution: Debt is only distributed when `WETH` is available, and the calculation uses `wethRewardPerVaultShare - lastVaultDistributedWethRewardPerShareX18`.
3. Accumulated Debt: If `WETH` is not available initially, debt will not be distributed until `WETH` becomes available. When it does, the system will attempt to distribute all debt from the begging, leading to incorrect calculations.

Impact

1. Incorrect Debt Accumulation: Debt for new vaults will not be accumulated until `WETH` is available for distribution.
2. Wrong `totalCreditCapacity`: The `totalCreditCapacity` of vaults will be computed incorrectly, leading to inaccurate debt calculations.
3. Unfair Distribution: Vaults added later may receive an unfair share of rewards due to the accumulation of WETH from the beginning of the market.

Tools Used

Manual Review

Recommendations

To fix this issue, ensure that the `CreditDelegation` struct is correctly initialized when a new vault is added. Specifically, set `lastVaultDistributedWethRewardPerShareX18` to the current `wethRewardPerVaultShare` at the time of vault creation. This will prevent the system from attempting to distribute all accumulated WETH from the beginning of the market.

[MEDIUM-03] `distributeProtocolAssetReward` Can DoS The Protocol

Summary

The `distributeProtocolAssetReward` function is designed to distribute protocol rewards to fee recipients. However, a critical issue arises when rewards are distributed in USDC during the `_convertAssetsToUsdc` and `_convertUsdcToAssets` processes, which are invoked within the `settleVaultsDebt` function based on the vault's state. If any fee recipient is blacklisted, both `distributeProtocolAssetReward` and `settleVaultsDebt` will revert, causing a denial-of-service (DOS) condition for the protocol.

Vulnerability Details

Consider the following scenario:

1. A keeper calls the `settleVaultsDebt` function.
2. The `distributeProtocolAssetReward` function is triggered, attempting to distribute rewards in USDC.
3. One of the fee recipients is blacklisted and cannot receive USDC rewards.
4. As a result, `distributeProtocolAssetReward` reverts, preventing the keeper from successfully executing `settleVaultsDebt`.

This issue stems from the protocol's reliance on a "push" mechanism for reward distribution, where rewards are directly sent to recipients. If any recipient is blacklisted, the entire transaction fails, halting the settlement process.

Impact

The vulnerability leads to a protocol-wide DOS condition. Until the blacklisted fee recipient is removed by the owner, the `settleVaultsDebt` function cannot be executed. Consequently, the protocol is unable to settle vault debts or credits, disrupting normal operations and potentially causing financial losses or inefficiencies.

Tools Used

Manual Review

Recommendations

To mitigate this vulnerability, the protocol should adopt a pull-over-push mechanism for reward distribution. Instead of directly sending rewards to fee recipients, the protocol should:

1. Store Rewards in a Separate Contract or Variable: Accumulate rewards in a designated storage variable or contract for each fee recipient.
2. Implement a Withdrawal Function: Allow fee recipients to withdraw their rewards at their discretion via a dedicated function (e.g., `withdrawRewards`).
3. Handle Blacklisted Recipients Gracefully: If a recipient is blacklisted, their rewards can remain in the protocol until they are no longer blacklisted or until the owner manually intervenes.

[MEDIUM-04] Missing Recalculate In `depositCreditForMarket`

Summary

In the current implementation of `depositCreditForMarket`, there is a missing call to `recalculateVaultsCreditCapacity`. This omission results in stale values for `totalDelegatedCreditUsd`, leading to incorrect calculations of `usdcCreditPerVaultShare`. Since the vaults of a market can accumulate more or less delegated credit over time, failing to update `totalDelegatedCreditUsd` before using it introduces inaccuracies.

Vulnerability Details

Scenario:

1. A user calls `depositCreditForMarket` with USDC as the deposit asset.
2. The function attempts to calculate `usdcCreditPerVaultShare` based on `totalDelegatedCreditUsd`.
3. However, `totalDelegatedCreditUsd` has not been updated—it may not reflect the latest state of the vaults.
4. This results in an incorrect credit per share calculation, which can impact credit distribution logic.

Impact

- Incorrect Credit Accounting: Since `usdcCreditPerVaultShare` relies on `totalDelegatedCreditUsd`, using stale values can distort credit distribution across vaults.
- Potential Financial Imbalance: Some vaults may receive more or less credit than they should

Tools Used

Manual Review

Recommendations

To ensure `totalDelegatedCreditUsd` is up to date before computing `usdcCreditPerVaultShare`, insert a call to `Vault.recalculateVaultsCreditCapacity()` at the beginning of `depositCreditForMarket`:

```
function depositCreditForMarket(
        uint128 marketId,
        address collateralAddr,
        uint256 amount
    )
        external
        onlyRegisteredEngine(marketId)
    {
        if (amount == 0) revert Errors.ZeroInput("amount");
```

```

+.    Vault.recalculateVaultsCreditCapacity(Market.loadLive(marketId).getConnectedVaultsIds());

        // loads the collateral's data storage pointer, must be enabled
        Collateral.Data storage collateral = Collateral.load(collateralAddr);
        collateral.verifyIsEnabled();
```

This change ensures that `totalDelegatedCreditUsd` always reflects the latest vault state, preventing incorrect credit calculations.

[MEDIUM-05] Stuck Funds

Summary

In `StabilityBranch` swap requests are first initiated through `initiateSwap` and then either fulfilled through `fulfillSwap` or refunded through `refundSwap`. For both fulfilling and refunding there is a base fee which is paid by the user - when fulfilled through the out amount and when refunded through the in amount. The problem is that a swap request can be initiated with such amount that cannot be either fulfilled or refunded due to a revert.

Vulnerability Details

In `StabilityBranch` swap requests are first initiated through `initiateSwap` and then either fulfilled through `fulfillSwap` or refunded through `refundSwap`. For both fulfilling and refunding there is a base fee which is paid by the user - when fulfilled through the out amount and when refunded through the in amount. The problem is that a swap request can be initiated with such amount that cannot be either fulfilled or refunded due to a revert. This happens because the base fee does not rely on the amount. If such a swap is created by a user (the system will let him to do so), his funds will be stuck without a way to get them back.

Impact

Stuck funds and loss of user trust.

Tools Used

Manual Review

Recommendations

Either change the way the base fee is calculated (for example as a percentage of the amount) or create a function that can be called only by the admin/owner to restore stuck funds from the contract.

[MEDIUM-06] `rebalanceVaultAssets` Will Not Work Correctly

Summary

During the `rebalanceVaultsAssets` function, there is a mechanism to swap underlying vault assets for USDC. However, an issue exists where `getExpectedOutput` returns an estimated value of funds needed from vault assets to obtain an exact amount of USDC. This estimate does not account for the final result of `executeSwapExactInputSingle`, leading to potential inconsistencies.

Vulnerability Details

The current implementation does not store the actual output of `executeSwapExactInputSingle` . Instead, it assumes the expected value is correct. Since slippage and liquidity variations can cause deviations, failing to use the actual value can lead to incorrect updates in the vaults' financial state.

Impact

If there is a discrepancy between the estimated and actual output values, the contract may not have sufficient funds to cover the changes. This could result in miscalculations in the vaults' balances, potentially leading to financial losses or failure in executing settlements.

Recommendations

To mitigate this issue, refactor the code to capture the actual swapped amount before proceeding with further calculations.

```
IERC20(ctx.inDebtVaultCollateralAsset).approve(ctx.dexAdapter, assetInputNative);
depositAmountUsdX18 = dexSwapStrategy.executeSwapExactInputSingle(swapCallData); // Capture actual swapped amount

uint128 usdDelta = depositAmountUsdX18.intoUint256().toUint128();
```

[MEDIUM-07] All Markets For A Vault Are Having The Same Weight

Summary

In the `recalculateVaultsCreditCapacity` function, the logic assigns the same weight to all markets using a vault. As a result, the entire `creditCapacity` of a vault is allocated to each market instead of being proportionally distributed. This leads to over-allocation, which can cause financial inconsistencies.

Affected Code

```
function updateVaultAndCreditDelegationWeight(
        Data storage self,
        uint128[] memory connectedMarketsIdsCache // Here, passing just the length would be sufficient
    )
        internal
    {
        // Cache the length of connected markets
        uint256 connectedMarketsConfigLength = self.connectedMarkets.length;

        // Load the connected markets storage pointer from the last configured market set
        EnumerableSet.UintSet storage connectedMarkets = self.connectedMarkets[connectedMarketsConfigLength - 1];

        // Get the total shares
        uint128 newWeight = uint128(IERC4626(self.indexToken).
   totalAssets());

        for (uint256 i; i < connectedMarketsIdsCache.length; i++) {
            // Load the credit delegation for the given market ID
            CreditDelegation.Data storage creditDelegation =
                CreditDelegation.load(self.id, connectedMarkets.at(i).
   toUint128());

            // Assign the entire newWeight to each market
            creditDelegation.weight = newWeight;
        }

        // Update the total vault weight
        self.totalCreditDelegationWeight = newWeight;
    }
```

Vulnerability Details

Scenario: Over-Allocation of Credit Capacity

1. A vault is connected to two markets.
2. Each market is assigned a weight of 1000, making the total weight also 1000.
3. If the vault has a `creditCapacity` of 100, it should distribute 50 to each market.
4. However, due to the current logic, each market receives the full 100, leading to over-allocation.

Impact

Markets will incorrectly assume that they have sufficient creditCapacity, when in reality, the total credit has been over-allocated across multiple markets. This can lead to:

Misleading credit availability – Markets may operate under the false assumption that they have enough liquidity.

Tools Used

Manual Review

Recommendations

Refactor the logic:

```
function updateVaultAndCreditDelegationWeight(
        Data storage self,
        uint128[] memory connectedMarketsIdsCache // Here, passing just the length would be sufficient
    )
        internal
    {
        // Cache the length of connected markets
        uint256 connectedMarketsConfigLength = self.connectedMarkets.length;

        // Load the connected markets storage pointer from the last configured market set
        EnumerableSet.UintSet storage connectedMarkets = self.connectedMarkets[connectedMarketsConfigLength - 1];

        // Get the total shares
        uint128 newWeight = uint128(IERC4626(self.indexToken).totalAssets());

        for (uint256 i; i < connectedMarketsIdsCache.length; i++) {
            // Load the credit delegation for the given market ID
            CreditDelegation.Data storage creditDelegation =
                CreditDelegation.load(self.id, connectedMarkets.at(i).toUint128());

            // Assign the entire newWeight to each market
            creditDelegation.weight = newWeight;
        }

        // Update the total vault weight
+        self.totalCreditDelegationWeight = newWeight * connectedMarketsIdsCache*length;
-        self.totalCreditDelegationWeight = newWeight;
    }
```

[MEDIUM-08] Configuring Fee Recipient Can Revert Wrongfully

Summary

In `MarketMakingEngineConfigurationBranch` there is a function `configureFeeRecipient` used to configure the fee recipient and their share. The fee preventing the fee being over the max configurable fee is wrong and can lead to reverts even when lowering the recipient's shares.

Vulnerability Details

In `MarketMakingEngineConfigurationBranch` there is a function `configureFeeRecipient` used to configure the fee recipient and their share. There is a check trying to prevent the fee being over the max configurable fee:

```
if (share > 0) {
    UD60x18 totalFeeRecipientsSharesX18 = ud60x18(marketMakingEngineConfiguration.totalFeeRecipientsShares);

    if (
        totalFeeRecipientsSharesX18.add(ud60x18(share)).gt(
            ud60x18(Constants.MAX_CONFIGURABLE_PROTOCOL_FEE_SHARES)
        )
    ) {
        revert Errors.FeeRecipientShareExceedsLimit();
    }
}
```

However, this check is wrong as it can revert even when trying to lower a recipient's share.
Let's have the following situation:

1. Recipient A's fee share is 0.9e18 (this is the maximum).
2. Owner tries to lower it to 0.7e18.
3. The check adds 0.9e18 to 0.7e18 and enforces that the result is less than 0.9e18 -> the call reverts.
4. Recipient A's fee share cannot be lowered.

Impact

Wrong reverted calls and non working max configurable fee logic.

Tools Used

Manual Review

Recommendations

Move this check after modifying the shares of the recipient and revert if the already changed `totalFeeRecipientsShares` is greater than the allowed maximum.

Low

[LOW-01] `fulfillSwap` And `initiateSwap` Should Execute `recalculateVaultsCreditCapacity`

Summary

The `fulfillSwap` function is designed to convert a specified USD amount into a collateral asset. However, the underlying logic in `getAmountOfAssetOut`, which calculates the discount for the swap, relies on potentially stale data. This is because the `recalculateVaultsCreditCapacity` function is not invoked before the calculation, leading to incorrect results.

Vulnerability Details

The vulnerability stems from the following code:

```
UD60x18 vaultAssetsUsdX18 = ud60x18(IERC4626(vault.indexToken).totalAssets()).mul(indexPriceX18); // audit bug
if (vaultAssetsUsdX18.isZero()) revert Errors.InsufficientVaultBalance(vaultId, 0, 0);

// We use the vault's net sum of all debt types coming from its
```

```
connected markets to determine the swap rate
SD59x18 vaultDebtUsdX18 = vault.getTotalDebt();

// Calculate the premium or discount that may be applied to the vault asset's index price
// Note: If no premium or discount needs to be applied, the premiumDiscountFactorX18 will be 1e18 (UD60x18 one value)
UD60x18 premiumDiscountFactorX18 =
     UsdTokenSwapConfig.load().getPremiumDiscountFactor(vaultAssetsUsdX18, vaultDebtUsdX18);

// Get amounts out, taking into consideration the CL price and the premium/discount
amountOutX18 = usdAmountInX18.div(indexPriceX18).mul(premiumDiscountFactorX18);
```

The issue lies in the fact that `getPremiumDiscountFactor` relies on `vaultAssetsUsdX18` and `vaultDebtUsdX18` to compute the premium or discount. However, these values may be stale if `recalculateVaultsCreditCapacity` is not called beforehand. This can lead to incorrect calculations of the `premiumDiscountFactorX18`, as the vault's debt and asset values may not reflect the current state.

Impact

The failure to invoke `recalculateVaultsCreditCapacity` before computing the `premiumDiscountFactorX18` can result in incorrect premium or discount values. This is particularly problematic in scenarios where the vault has excessive debt or credit. If the vault's debt is understated due to stale data, a lower premium factor may be applied, leading to an unfavorable swap rate. This could harm the vault by allowing more assets to be swapped than intended, potentially resulting in financial losses.

Tools Used

Manual Review

Recommendations

To address this issue, refactor the code to ensure that `recalculateVaultsCreditCapacity` is called before calculating the `premiumDiscountFactorX18`. Here's an example of how the code can be updated:

```
ctx.vaultId = request.vaultId;
Vault.Data storage vault = Vault.loadLive(ctx.vaultId);

+ recalculateVaultsCreditCapacity(vaultId);
```

Same fix should be added to the `initiateSwap`, so the minAmount check to be applied correctly

[LOW-02] Swap Requests Can Be Initiated For Vaults That Are Not Live

Summary

Currently swap requests follow the given workflow:

1. A swap request is initiated through `initiateSwap`
2. After this there are two choices:
    - Fulfil request through `fulfillSwap`
    - Refund request through `refundSwap`

The problem is that a swap can be initiated for a vault that is not live but can be fulfilled only for a vault that is live leaving the user forced to refund the request and pay fees.

Vulnerability Details

Swap requests can be initiated with vaults that are not live which will lead to the user being forced to pay fee on refund and not be able to fulfil the swap. This is due to the vault being loaded using `load` in `initiateSwap` but `loadLive` in `fulfillSwap`.

Impact

Users paying fee for a request that was never possible in the concrete situation. This will lead to trust issues with protocol.

Tools Used

Manual Review

Recommendations

Add a check whether the given vault is live when a user is initiating a swap request and revert if it is not.