Title: Zaros Part 2

Author: 0xshrxyeh
Date: January 6, 2025

Prepared by: 0xshrxyeh

Disclaimer

Shreyash Naik makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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.

Table of contents

Contest Summary

Sponsor: Zaros

Dates: Jan 20th, 2025 - Feb 6th, 2025

See more contest details here

Results Summary

Number of findings:

- High: 5
- Medium: 15
- Low: 15

High Risk Findings

H-01. Engine Parameter Mismatch in StabilityBranch::fulfillSwap

Summary

In the `fulfillSwap` function, the keeper-provided parameter `address engine` is used to set the stablecoin token (usdToken) from the configuration without verifying that the passed engine matches the engine associated with the vault. This can lead to using an incorrect USD token and cause unexpected state changes.

Vulnerability Details

In the current implementation the function:

- Accepts priceData from an external keeper without verifying that it is intended for the current blockchain.
- Does not verify any contextual identifiers (like chain id, nonce, or unique request id) that bind the data to a specific deployment.
- This creates a risk that a keeper could replay the same off-chain price data on different chains (in a multi-chain environment) allowing unauthorized or double fulfillment of swap requests.
- For example, the pseudo-code below shows the basic logic of the problematic function:

```
function fulfillSwap(user, requestId, priceData, engine) {
    // Modifier ensures only registered system keepers can invoke this function.
    require(isRegisteredSystemKeeper(msg.sender), "Unauthorized");

    // The price data is verified against off-chain sources,
    // however it does not include any chain-specific identifier.
    let verifiedPrice = verifyPriceData(priceData); // Missing chain-ID check!

    // Engine parameter is provided from the caller with no further validation.
    if (engine ! = expectedEngineAddress) {
        throw new Error("Engine parameter mismatch");
    }

    // Further logic completes the swap without binding priceData to the current ch
    // ...
}
```

Impact

An attacker or misconfigured keeper operating in a cross-chain scenario could replay the same price data on a different chain. This could lead to multiple approvals of the same swap request, causing incorrect asset transfers, double fulfillment, or other systemic inconsistencies.aq

Tools Used

- Manual code review
- Static analysis (e.g., Slither, MythX)
- Fuzz testing

Recommendations

- **Parameter Verification:** Rigorously validate the `engine` parameter against known registered engines in the system.
- **Tight Access Controls:** Use internal mappings or require additional context (e.g., linking vault IDs with the expected engine) to enforce correctness.
- **Nonce or Timestamp Checks:** Require a nonce or time-bound parameter in the price data to ensure one-time usage.
- **Testing:** Implement unit tests that simulate incorrect engine inputs.

H-02. Price manipulation vulnerability in swap calculations in StabilityBranch.sol::getAmountOfAssetOut

Summary

Swap rate calculations are based on the vault's total assets and debt without implementing sanity checks on the premium/discount factor. An attacker could manipulate the vault's asset balance (e.g., via flash loans) immediately before swap execution to force extreme swap rates.

Vulnerability Details

The function `getAmountOfAssetOut` computes the swap output as follows:

```
 UD60x18 vaultAssetsUsdX18 =
ud60x18(IERC4626(vault.indexToken).totalAssets()).mul(indexPriceX18);
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();
```

POC

```
function getAmountOfAssetOut(vaultId, usdAmountIn, indexPrice) {
  let vault = loadVault(vaultId);
  let vaultAssetsUsd = totalAssets(vault.indexToken) * indexPrice;
  if (vaultAssetsUsd === 0) throw Error("InsufficientVaultBalance");

  let vaultDebt = vault.getTotalDebt();
  let premiumDiscount = getPremiumDiscountFactor(vaultAssetsUsd, vaultDebt);
```

```
  // The computed output may be exaggerated if vaultAssetsUsd is manipulated.
  let amountOut = (usdAmountIn / indexPrice) * premiumDiscount;
  return amountOut;
}
```

Impact

- **Exploitable Swap Rates:** An attacker may temporarily lower the vault's total asset value, forcing a severe discount or premium, to gain an advantageous swap rate.
- **Financial Losses:** Manipulated rates could lead to draining a vault's reserves or unfairly benefiting malicious parties.

Tools Used

- Static analysis (Slither, MythX)
- Manual review and economic simulation
- Fuzz testing

Recommendations

- **Rate Limits & Sanity Checks:** Impose minimum/maximum bounds on the computed premium/discount factor.
- **Time-weighted Oracle Values:** Use time-averaged or oracle-verified asset values rather than instantaneous totalAssets() calls.
- **Flash Loan Guards:** Introduce measures to detect or block flash loan manipulations of vault balances.

H-03. Liquidated Accounts in Profit Overcharged Due to Absolute Unrealized PnL in LiquidationBranch.sol::liquidateAccounts()

Summary

In the liquidation process, the protocol uses the absolute value of the account's unrealized PnL when deducting margin. This causes accounts that are actually in profit to be overcharged—effectively confiscating their positive PnL—because the liquidation mechanism treats gains as losses.

Vulnerability Details

Inside the `liquidateAccounts(...)` function, the margin to deduct is calculated by adding the required maintenance margin to the absolute value of the unrealized PnL. Specifically, the snippet:

```
 ctx.liquidatedCollateralUsdX18 = tradingAccount.deductAccountMargin(
TradingAccount.DeductAccountMarginParams({feeRecipients: FeeRecipients.Data({margin
perpsEngineConfiguration.marginCollateralRecipient,
orderFeeRecipient: address(0),
settlementFeeRecipient:
```

```
perpsEngineConfiguration.liquidationFeeRecipient}),
pnlUsdX18:
ctx.accountTotalUnrealizedPnlUsdX18.abs().intoUD60x18().add(ctx.requiredMaintenance
orderFeeUsdX18: UD60x18_ZERO,settlementFeeUsdX18: ctx.liquidationFeeUsdX18,marketId
accountPositionsNotionalValueX18:ctx.accountPositionsNotionalValueX18}));
```

Impact

- **Financial Loss for Users**
- **Unfair Liquidation**
- **Reputational Damage**

Tools Used

- **Manual Code Review:** A detailed inspection of the liquidation logic in `LiquidationBranch.sol`.
- **Static Analysis Tools:** Identification and flagging of the misuse of the `abs()` function in financial computations.
- **Unit Testing & Simulation:** Simulated liquidation scenarios were used to evaluate how the margin deduction behaves when the unrealized PnL is positive.

Recommendations

- **Conditional PnL Handling:** Modify the margin deduction logic so that it conditionally handles PnL:
  - Apply the absolute value only for negative (loss) scenarios.
  - For positive PnL, do not add the profit to the maintenance margin, allowing the profit to offset the margin requirement.
- **Reevaluate Margin Calculations:** Review the liquidation requirements to ensure that the calculated margin properly reflects both losses and gains, avoiding the confiscation of positive PnL.
- **Implement Comprehensive Tests:** Introduce unit and integration tests that cover both profit and loss cases during liquidation to validate the corrected logic under various market conditions.
- **Peer Audit:** Once changes are implemented, have an external audit or peer review to verify that the new mechanism correctly handles the nuances of profit and loss during liquidation.

H-04. Potential for Sandwich Attacks During Liquidation in LiquidationBranch.sol::liquidateAccounts

Summary

The liquidation process calculates the mark price at execution time without incorporating any slippage or price manipulation protection. This can allow malicious actors to sandwich the liquidation by manipulating the market price before and after the liquidation call, potentially causing the liquidated trader to suffer larger losses.

Vulnerability Details

In the liquidation workflow within the `liquidateAccounts` function, the following operations are performed:

```
 // calculate price impact of open position being closed
ctx.markPriceX18 = perpMarket.getMarkPrice(ctx.liquidationSizeX18, perpMarket.getIn
// calculate notional value of the position being liquidated
ctx.accountPositionsNotionalValueX18[j] = ctx.oldPositionSizeX18.abs().intoUD60x18(
```

Impact

- **Increased Losses for Liquidated Traders:** A manipulated mark price may force liquidations at adverse prices, stripping more collateral from traders than necessary.
- **Exploitation by Malicious Actors:** Attackers could profit from adverse price moves generated specifically to trigger liquidations under manipulated conditions.
- **Erosion of Market Fairness:** Repeated exploitation could undermine the trust in the protocol, making users wary of entering positions due to manipulation risks.

Tools Used

- **Manual Code Review**
- **Static Analysis**

Recommendations

- **Implement Slippage Controls:** Introduce a mechanism to define a maximum acceptable deviation from the expected mark price. If the deviation exceeds a certain threshold, the liquidation should be aborted or adjusted.
- **Utilize Robust Price Oracles:** Consider integrating a time-weighted average price (TWAP) oracle or multiple price feeds to mitigate sudden market manipulation.
- **Add Price Verification:** Introduce a secondary price check within a short delay or multi-step confirmation process before executing liquidation orders.
- **Deploy Circuit Breakers:** Consider implementing a circuit breaker that pauses liquidations when market volatility exceeds predefined safety limits.

H-05. Lack of Slippage Protection for Liquidation Orders in LiquidationBranch.sol::liquidateAccounts()

Summary

The liquidation process does not include slippage protection when calculating the mark price for closing positions. The price is derived directly from market data at runtime with no safeguards to check if it deviates significantly from expected values. This omission exposes the protocol to unfavorable liquidations and potential market manipulation.

Vulnerability Details

Within the `liquidateAccounts()` function, the mark price is calculated as follows:

```
ctx.markPriceX18 = perpMarket.getMarkPrice(ctx.liquidationSizeX18, perpMarket.getIr
```

Impact

- **Excessive Liquidation Losses:** Traders may suffer greater losses as the liquidation occurs at manipulated or unfavorable prices.
- **Market Manipulation Risk:** Malicious actors could exploit this vulnerability to intentionally manipulate prices, triggering liquidations that benefit them at the expense of liquidated traders.
- **Protocol Integrity:** Repeated exploitation could undermine trust in the protocol and discourage participation in the trading platform.

Tools Used

- **Manual Code Review**
- **Static Analysis Tools**
- **Simulation Testing**

Recommendations

- **Implement Slippage Limits:** Introduce parameters to define a maximum acceptable deviation from an expected mark price. Abort or adjust liquidations if the deviation exceeds this threshold.
- **Integrate Robust Price Feeds:** Use time-weighted average prices (TWAP) or multiple price oracles to mitigate the impact of sudden, short-term price fluctuations.
- **Add Price Verification Steps:** Consider incorporating a confirmation mechanism or secondary price check before finalizing liquidations.
- **Document Expected Behavior:** Clearly document the acceptable slippage range and expected behavior under volatile conditions, ensuring that both the development team and users understand the risk mitigation measures.

Medium Risk Findings

M-01. Unrestricted approvals in FeeDistributionBranch.sol::_performMultiDexSwap()

Summary

The function _performMultiDexSwap(...) repeatedly approves tokens without resetting allowances, which could lead to potential misuse by a malicious or compromised DEX adapter.

Vulnerability Details

- The function approves tokens for DEX adapters without resetting these allowances afterward.

- A malicious adapter could potentially move more funds than intended if it is untrusted or if there is a bug allowing calls under different parameters.

javascript

```
IERC20(assets[i]).approve(dexSwapStrategy.dexAdapter, amountIn);
...
amountIn = dexSwapStrategy.executeSwapExactInputSingle(swapCallData);
```

Impact

If an adapter is compromised or untrusted, it could exploit the unrestricted approvals to transfer more tokens than intended. This could result in unauthorized token drains.

Tools Used

Manual code review

Recommendations

- Implement a mechanism to reset token allowances after swaps.
- Ensure that only trusted DEX adapters are used.

M-02. Lack of DEX Return Amount Validation in FeeDistributionBranch::convertAccumulatedFeesToWeth function

Summary

The function `convertAccumulatedFeesToWeth` does not validate the minimum output amount during swaps, exposing the protocol to slippage and sandwich attacks.

Vulnerability Details

- The function performs swaps without checking if the output meets a minimum threshold.
- This could result in significant value loss due to slippage or sandwich attacks.

javascript

```
if (path.length == 0) {
SwapExactInputSinglePayload memory swapCallData =
SwapExactInputSinglePayload({
tokenIn: asset,
tokenOut: ctx.weth,
amountIn: ctx.assetAmount,
recipient: address(this)
});
```

```
ctx.tokensSwapped =dexSwapStrategy.executeSwapExactInputSingle(swapCallData);
}
```

Impact

Potential loss of value when converting accumulated fees to WETH,
especially in volatile or low liquidity market conditions.

Tools Used

Manual code review

Recommendations

- Implement minimum output amount checks to guard against unfavorable slippage.
- Consider using a slippage tolerance parameter.

M-03. No Slippage Control in WETH Conversions in FeeDistributionBranch::convertAccumulatedFeesToWeth

Summary

There is no mechanism to guard against unfavorable slippage, potentially exposing the protocol to front-running. A malicious party can anticipate a large token-to-WETH trade, adjust the on-chain liquidity or manipulate the price, and force the protocol to swap at a notably worse rate. The code does not verify that the final output meets a minimum threshold, which can result in significantly reduced WETH proceeds for the protocol and vault participants.

Vulnerability Details

- The convertAccumulatedFeesToWeth function lacks slippage control mechanisms, enabling front-running attacks.
- The function does not verify that the final output meets a minimum threshold.
- Malicious parties can manipulate on-chain liquidity or prices to force swaps at worse rates.

```
 function convertAccumulatedFeesToWeth(
uint128 marketId,
address asset,
uint128 dexSwapStrategyId,
bytes calldata path
) external onlyRegisteredSystemKeepers {
// ...calls dexSwapStrategy.executeSwapExactInput or executeSwapExactInputSingle...
}
```

Impact

This vulnerability can lead to significantly reduced WETH proceeds, harming both the protocol and its participants.

Tools Used

Manual code review

Recommendations

- Enforce slippage limits by integrating checks for acceptable price ranges.
- Set minimum acceptable output amounts for swaps.

M-04. Rounding Errors in FeeDistribution.sol::receiveMarketFee

Summary

The function receiveMarketFee(...) does not account for fee-on-transfer tokens, leading to discrepancies between expected and actual token amounts

Vulnerability Details

- The function assumes the received token amount equals the transferred amount.
- Fee-on-transfer tokens result in fewer tokens being received than expected.

```
//The function receiveMarketFee(...) uses:

IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
```

to receive tokens assumed to be equal to amount . However, tokens that charge transfer fees (often called fee-on-transfer) would result in the contract receiving fewer tokens than expected. Because the contract records the entire amount in its internal ledger, there is a discrepancy between the real token amount and the accounted amount. This discrepancy can lead to inflated tracked balances and unexpected shortfalls when converting accumulated fees to WETH.

Impact

Inflated tracked balances and unexpected shortfalls when converting fees to WETH.

Tools Used

Manual code review.

Recommendations

- Adjust the recorded amount to reflect the actual received tokens.
- Implement checks for fee-on-transfer tokens.

M-05. Unenforced deadline checks in StabilityBranch.sol::fulfillSwap

Summary

The `fulfillSwap` function checks the swap request's deadline and then marks the request as processed. However, there is a potential race condition: after passing the deadline check but before swap execution, delays can occur that effectively execute swaps post-deadline.

Vulnerability Details

```
 // if request dealine expired revert
ctx.deadline = request.deadline;
if (ctx.deadline < block.timestamp) {
revert Errors.SwapRequestExpired(user, requestId, ctx.deadline);
}
// set request processed to true
request.processed = true;
```

POC

```
function fulfillSwap(request) {
  // Check if the deadline has passed.
  if (request.deadline < currentTimestamp()) {
    throw Error("SwapRequestExpired");
  }

  // Immediately mark the request as processed.
  request.processed = true;

  // Further delay in external calls or complex processing could allow execution pa
  executeSwapLogic();
}
```

Impact

- **Delayed Execution:** A malicious keeper could intentionally delay the final swap execution after the deadline.
- **User Disadvantage:** Users lose the opportunity to cancel or reclaim funds if the deadline is effectively bypassed by delays.

Tools Used

- Manual review

- Time-based simulation tests
- Fuzz testing (Forge)

Recommendations

- **Reordering Operations:** Perform all time-sensitive actions (or recheck the deadline) immediately before transferring funds.
- **Immediate Effects:** Mark requests only after all external calls succeed, using a design that minimizes delay.
- **Timestamp Revalidation:** Consider revalidating the deadline after critical state updates.

M-06. Unbounded array inputs in StabilityBranch.sol::initiateSwap

Summary

The `initiateSwap` function accepts three array parameters (`vaultIds` , `amountsIn` , and `minAmountsOut`) without imposing a maximum size limit. While the function checks that all arrays have matching lengths, it does not restrict the number of swap requests that can be submitted in a single transaction. This opens the door to potential Denial of Service (DoS) attacks where a malicious user could supply a massive batch of swaps, leading to excessive gas consumption and possibly exceeding block gas limits.

Vulnerability Details

The function is defined as:

```
 function initiateSwap(
uint128[] calldata vaultIds,
uint128[] calldata amountsIn,
uint128[] calldata minAmountsOut
) external
```

##POC

```
 function initiateSwap(vaultIds, amountsIn, minAmountsOut) {
// Verify that all arrays are of the same length.
if (vaultIds.length ! = amountsIn.length || amountsIn.length !== minAmountsOut.leng
throw new Error("ArrayLengthMismatch");
}
// Process each swap request (this loop can iterate an unbounded number of times).
for (let i = 0; i < vaultIds.length; i++) {
// Each iteration involves multiple storage accesses and external interactions.
processSwap(vaultIds[i], amountsIn[i], minAmountsOut[i]);
}
}
```

Impact

- **Denial of Service (DoS):** An attacker can force the transaction to consume an excessive amount of gas by supplying very large arrays, causing the transaction to exceed the block gas limit.
- **Resource Exhaustion:** The excessive gas consumption and numerous storage operations could lead to network congestion or degraded performance for legitimate users.
- **Blocked Operations:** Legitimate swap operations could be prevented from execution if the network state is overwhelmed by such resource-intensive transactions.

Tools Used

- **Manual Code Review:** Careful inspection of the array input validations.
- **Static Analysis Tools:** Tools such as Slither and MythX helped identify the lack of an upper limit.
- **Fuzz Testing:** Frameworks like Forge Fuzz Testing simulated inputs with large array sizes to assess gas consumption.

Recommendations

- **Introduce an Upper Bound:** Enforce a maximum allowed length for the input arrays. For example, add a check such as:

```
if (vaultIds.length > MAX_BATCH_SIZE) {
    revert Errors.ArrayLengthExceedsLimit(vaultIds.length, MAX_BATCH_SIZE);
}
```

- **Batch Processing:** Consider splitting the processing into smaller batches if large numbers of swaps need to be supported.
- **Gas Profiling:** Regularly profile the gas consumption for this function and adjust the limits accordingly.
- **Input Validation:** Add input validation to reject unusually large arrays at the contract level before commencing any processing.

M-07. Missing deadline validation in StabilityBranch.sol::initiateSwap

Summary

The `initiateSwap` function does not allow users to set their own transaction deadline. Instead, it relies on a fixed `maxExecutionTime` from the contract configuration. This forces all users to accept the same execution delay regardless of their risk tolerance or market conditions, reducing user control over transaction timing.

Vulnerability Details

The function sets the deadline using a contract-wide `maxExecutionTime` rather than a user-specified parameter:

```
function initiateSwap(
uint128[] calldata vaultIds,
uint128[] calldata amountsIn,
uint128[] calldata minAmountsOut
) external {
```

```
// ...
ctx.maxExecTime = uint120(tokenSwapData.maxExecutionTime);
// ...
ctx.deadlineCache = uint120(block.timestamp) + ctx.maxExecTime;
swapRequest.deadline = ctx.deadlineCache;
}
```

Impact

- **Reduced User Control:** Users cannot specify a maximum acceptable execution time, forcing them to accept a one-size-fits-all deadline.
- **Excessive Pending Time:** If the global `maxExecutionTime` is too long, users may experience increased uncertainty and risks during high volatility.
- **Exploitation by MEV Bots:** Bots may time their transactions to execute near the end of the global deadline period to exploit price movements.
- **Market Conditions Mismatch:** In volatile market conditions, users might prefer shorter deadlines to mitigate risk, which is not possible under the current scheme.

Tools Used

- **Manual Code Review:** Detailed inspection of the contract code revealed the fixed deadline approach.
- **Static Analysis Tools:** Tools like Slither and MythX were used to analyze the control flow and parameter handling.
- **Fuzz Testing:** Tests were run using frameworks (e.g., Forge) to simulate different user input scenarios and assess the inflexibility in deadline specification.

Recommendations

- **User-Specified Deadline Parameter:** Modify the `initiateSwap` function to accept an additional `deadline` parameter from the user. This allows users to set their own acceptable transaction execution window.

  For example:

```
 function initiateSwap(
    uint128[] calldata vaultIds,
    uint128[] calldata amountsIn,
    uint128[] calldata minAmountsOut,
    uint120 deadline  // New parameter provided by the user
) external {
    // Compare the provided deadline against a minimum or maximum allowable time
    if (deadline < block.timestamp + MIN_EXECUTION_TIME || deadline > block.times
        revert Errors.InvalidDeadline(deadline);
    }

    // Use the provided deadline instead of a fixed global configuration.
    swapRequest.deadline = deadline;
```

```
    // Continue with swap processing...
}
```

- **Configurable Range Checks:** Enforce lower and upper bounds on the user-specified deadline to prevent extreme values.
- **User Interface Clarity:** Update the user interface to clearly explain the risks and benefits of choosing a custom deadline, allowing users to make informed decisions during volatile market conditions.
- **Testing and Simulation:** Rigorously test the new deadline feature under various market conditions to ensure that it behaves as intended without introducing additional risks.

M-08. Keeper Price Verification Lacks Deviation Checks in StabilityBranch.sol::fulfillSwap

Summary

The `fulfillSwap` function verifies keeper-provided off-chain price data using `verifyOffchainPrice()`, but it does not validate that the reported price falls within an expected deviation range compared to prior swaps or trusted price feeds. Without deviation checks, the system may accept extreme price changes within a single transaction, potentially leading to price manipulation attacks or acceptance of stale prices, exposing the protocol to economic risks.

Vulnerability Details

The current implementation obtains the price as follows:

```
function fulfillSwap(...) external {
// ...
// Price verification lacks deviation checks
ctx.priceX18 = stabilityConfiguration.verifyOffchainPrice(priceData);
ctx.amountOutBeforeFeesX18 = getAmountOfAssetOut(
ctx.vaultId,
ud60x18(ctx.amountIn),
ctx.priceX18
);
// ...
}
```

Impact

- **Price Manipulation:** Malicious or compromised keepers might supply a price with a drastic deviation, leading to unfair swap rates and potential asset loss.
- **Acceptance of Stale Prices:** In scenarios of network delay or flash crashes, stale prices might be used, exposing users to unexpected slippage.

- **Economic Exploits:** An attacker could benefit from intentionally triggering abnormal price movements within a single transaction, resulting in arbitrage opportunities or draining vault liquidity.

Tools Used

- **Manual Code Review:** Inspection of the price verification process in the `fulfillSwap` function.
- **Static Analysis:** Tools like Slither and MythX identified the absence of deviation checks.
- **Fuzz Testing:** Input variations simulated with frameworks like Forge demonstrated potential for abnormal price inputs.

Recommendations

- **Implement Deviation Checks:** Introduce logic to verify that the keeper-provided price does not vary beyond an acceptable range (e.g., a percentage spike) compared to a recent reference price or an oracle-based average.

  For example:

```
 uint256 allowedDeviationBps = 200; // Example: 2% deviation allowed.
uint256 referencePrice = getReferencePrice(); // Retrieve trusted, recent price.
uint256 lowerBound = referencePrice.mul(BPS_DENOMINATOR.sub(allowedDeviationBps))
uint256 upperBound = referencePrice.mul(BPS_DENOMINATOR.add(allowedDeviationBps))

if (priceX18 < lowerBound || priceX18 > upperBound) {
    revert Errors.PriceDeviationTooHigh(priceX18, referencePrice);
}
```

- **Integrate Additional Oracles:** Use multiple price feeds or a moving average to better understand normal market deviations.
- **Reporting and Alerts:** Implement monitoring to alert if price deviations occur frequently, suggesting potential manipulation or network issues.
- **Rigorous Testing:** Simulate abnormal price scenarios to ensure that the deviation check works as intended without impairing legitimate swap activity.

M-09. Cross-chain Replay Attacks in StabilityBranch.sol::fulfillSwap

Summary

The `fulfillSwap` function uses off-chain price data provided by keepers without binding this data to a specific chain. As a result, a price report verified on one chain could be replayed on another chain where the contract is deployed.

Vulnerability Details

The problematic verification workflow is as follows:

```
function fulfillSwap(user, requestId, priceData, engine) {
  // Verifies off-chain price data without checking chain-specific identifiers.
  let verifiedPrice = verifyPriceData(priceData); // No chainId or unique nonce che

  // Continues processing the swap based on the unbound price data.
  processSwap(verifiedPrice, ...);
}
```

Impact

- **Replay Exploits:** An attacker could replay verified price data from one chain to another, causing the same swap request to be executed or approved multiple times.
- **Double Spends:** This replay may lead to duplicate processing, asset misallocation, or unauthorized reward distribution, undermining the protocol's integrity.

Tools Used

- Cross-chain security analysis
- Static analysis (MythX, Slither)
- Manual code and design review

Recommendations

- **Chain-specific Data:** Embed chain identifiers (e.g., chain ID, network nonce) within the price data payload and enforce that the verified report comes from the intended chain.
- **Nonce Management:** Use nonces or time-bound tokens to ensure that price data is one-time use only.
- **Verification Updates:** Enhance the verification logic in the Chainlink verifier proxy to include chain-specific validations.

M-10. No Clear Redistribution of Leftover Margin After Liquidation in LiquidationBranch.sol::liquidateAccounts()

Summary

The liquidation process deducts the maintenance margin, liquidation fee, and a calculated unrealized PnL from the user's account without any explicit mechanism to return any leftover collateral. This behavior results in the complete depletion of the user's margin, even if excess funds remain that are not strictly required to cover the liquidation costs.

Vulnerability Details

Within the `liquidateAccounts(...)` function, the protocol calls `tradingAccount.deductAccountMargin` to remove the necessary funds from the user's collateral. The code snippet below illustrates this process:

```
ctx.liquidatedCollateralUsdX18 = tradingAccount.deductAccountMargin(
    TradingAccount.DeductAccountMarginParams({
        feeRecipients: FeeRecipients.Data({
            marginCollateralRecipient: perpsEngineConfiguration.marginCollateralRec
            orderFeeRecipient: address(0),
            settlementFeeRecipient: perpsEngineConfiguration.liquidationFeeRecipien
        }),
        pnlUsdX18: /* computed pnl value */,
        orderFeeUsdX18: UD60x18_ZERO,
        settlementFeeUsdX18: ctx.liquidationFeeUsdX18,
        marketIds: ctx.activeMarketsIds,
        accountPositionsNotionalValueX18: ctx.accountPositionsNotionalValueX18
    })
);
```

The lack of an obvious redistribution or refund mechanism for any remaining margin implies that, if the user had more collateral than the sum of the maintenance margin, fees, and perceived losses, the excess is not returned. This could essentially zero out the user's collateral even when they held a surplus.

Impact

- **User Fund Losses:** Liquidated traders might lose extra collateral beyond what is necessary to cover their liabilities, unfairly penalizing them.
- **Incentive Misalignment:** The absence of a refund mechanism for excess margin funds could reduce user confidence in the protocol, as traders may feel their funds are being taken unnecessarily.
- **Protocol Fairness:** Without clear rules on the redistribution of leftover funds, there can be ambiguity in margin accounting, leading to potential disputes and a lack of trust in the liquidation process.

Tools Used

- **Manual Code Review:** A thorough inspection of the liquidation logic in the `LiquidationBranch.sol` was conducted.
- **Static Analysis:** Automated tools were used to trace fund flows and detect discrepancies in collateral handling.
- **Integration Testing:** Test scripts simulating liquidation scenarios helped identify that excess margin was not being redistributed.

Recommendations

- **Implement a Refund Mechanism:** Adjust the liquidation logic to explicitly calculate and refund any leftover margin back to the liquidated user. This would involve:
  - Determining the exact surplus after deducting required amounts.
  - Safely crediting the surplus back to the user's account.
- **Enhance Documentation:** Clearly document the intended behavior regarding any surplus margin so that users understand what happens to their collateral during liquidation.

- **User Notifications:** Consider emitting additional events that detail the amount of leftover margin credited back to users for better transparency.
- **Thorough Testing:** Develop comprehensive tests to cover scenarios where the deducted margin is less than the total available funds, ensuring that the refund mechanism operates correctly without introducing new vulnerabilities.

M-11. Potential DoS through Large Input Array in LiquidationBranch.sol::liquidateAccounts()

Summary

The `liquidateAccounts()` function iterates over every account ID provided in the input array and performs multiple storage updates and complex operations for each account. If a malicious actor provides an excessively large array, the gas cost of processing the transaction may exceed the block gas limit, leading to out-of-gas errors and resulting in a denial-of-service (DoS) condition for legitimate liquidation attempts.

Vulnerability Details

The function is defined as follows:

```
function liquidateAccounts(uint128[] calldata accountsIds) external {
    // ...
    for (uint256 i; i < accountsIds.length; i++) {
        ctx.tradingAccountId = accountsIds[i];
        // ... heavy operations (clearing orders, updating positions, computing fun
    }
}
```

Since the function performs numerous storage operations and complex logic per iteration for each element in `accountsIds`, the gas consumption increases linearly with the length of the array. This makes the function susceptible to abuse if a very large list is provided. In such a scenario, the transaction may revert due to an out-of-gas condition, effectively preventing any liquidations within that block and potentially delaying the liquidation of underwater accounts.

Impact

- **Denial-of-Service (DoS):** Legitimate liquidators may be unable to complete their liquidation transactions in a timely manner if the function call consumes excessive gas, delaying the liquidation process.
- **Operational Disruption:** In a volatile market, delays in liquidation can result in increased risk and potential financial losses for the protocol and its traders.
- **Exploitation by Malicious Actors:** An attacker could purposely submit a very large input array to disrupt the liquidation process and gain an advantage in volatile trading scenarios.

Tools Used

- **Manual Code Review:** Inspection of the loop in `liquidateAccounts()` revealed the potential for high gas consumption.
- **Static Analysis:** Analysis tools flagged concerns over loops iterating over user-provided arrays without bound.
- **Simulation Testing:** Hypothetical scenarios were tested to estimate the gas cost impact of processing large arrays.

Recommendations

- **Input Array Length Limitation:** Implement a maximum length check for the `accountsIds` array at the start of the function to prevent excessive gas consumption.

```
require(accountsIds.length <= MAX_ALLOWED_ACCOUNTS, "Input array exceeds maximum
```

- **Batch Processing:** Consider splitting a large array into multiple smaller batches. Allow multiple calls to the function to process liquidations gradually rather than in a single transaction.
- **Optimized Processing:** Explore optimizations in the liquidation logic to reduce gas cost per iteration, such as off-chain pre-processing or more efficient storage updates.
- **Gas Refunds or Incentives:** Provide appropriate gas reimbursements or incentives for liquidators handling large datasets to mitigate the impact of higher gas costs.

M-12. In LiquidationBranch.sol::liquidateAccounts() there is lack of Skew and Open Interest Limit Enforcement During Liquidations

Summary

During liquidations, the protocol intentionally bypasses the enforcement of skew and open interest limits by passing `false` to the check function. Although the rationale is to protect against DoS attacks, this approach may allow the market's open interest and skew to reach unsafe levels under certain conditions, potentially leading to market instability and manipulation.

Vulnerability Details

In the liquidation process, the protocol executes the following code:

```
// we don't check skew during liquidations to protect from DoS
(ctx.newOpenInterestX18, ctx.newSkewX18) = perpMarket.checkOpenInterestLimits(ctx.l

// update perp market's open interest and skew
perpMarket.updateOpenInterest(ctx.newOpenInterestX18, ctx.newSkewX18);
```

Here, the enforcement parameter is set to `false`, effectively disabling strict limit checks during a liquidation event. While the assumption is that open interest and skew decrease during liquidations, there is no runtime validation to ensure these values do not reach dangerous levels. Under volatile or complex market conditions—especially if

multiple large positions are liquidated simultaneously—this lack of enforcement may lead to market conditions that are outside of safe operational parameters.

Impact

- **Market Instability:** Without active enforcement, skew and open interest may reach levels that could destabilize the market or trigger unintended risk cascades.
- **Manipulation Risks:** Attackers might exploit this gap by orchestrating liquidations that intentionally push these market parameters into unsafe ranges.
- **Risk Management Failure:** The absence of proper checks undermines market safeguards, posing systemic risk during periods of high volatility or clustered liquidation events.

Tools Used

- **Manual Code Review:** Analysis of the liquidation workflow identified the bypass of limit enforcement.
- **Static Analysis Tools:** Automated tools highlighted that the check for skew and open interest limits is not enforced during liquidation.
- **Risk Simulation:** Hypothetical scenarios indicated that mass liquidations without proper limits could misalign market parameters.

Recommendations

- **Reinstate Limit Enforcement:** Implement a conditional mechanism that enforces open interest and skew limits during liquidation, even if in a modified or optimized form. For example, instead of passing `false` outright, consider conditionally enforcing minimum limits based on market volatility.
- **Dynamic Safeguards:** Adjust the liquidation logic to incorporate dynamic checks that can scale with market conditions, ensuring that large liquidations trigger stricter validations.
- **Circuit Breakers:** Introduce safety measures or circuit breakers that pause liquidations if skew or open interest exceed predefined thresholds during volatile conditions.
- **Extensive Testing:** Develop rigorous simulations and integration tests to determine safe operating parameters for skew and open interest during liquidation scenarios.

M-13. Missing Time-Based Liquidation Cooldown Period in LiquidationBranch.sol::liquidateAccounts()

Summary

The liquidation mechanism does not enforce any time-based cooldown between successive liquidation attempts. As a result, accounts that have just been liquidated (or are borderline liquidatable) can be immediately targeted again. This absence of a grace period prevents traders from adding margin or adjusting positions, potentially leading to unnecessarily aggressive and repeated liquidations.

Vulnerability Details

In the `liquidateAccounts(uint128[] calldata accountsIds)` function, there is no check of a time-stamp or cooldown parameter to ensure that sufficient time has passed between liquidation attempts on the same account. For example, the function implementation:

```
function liquidateAccounts(uint128[] calldata accountsIds) external {
    // No check for last liquidation attempt timestamp
    // Immediate processing of liquidation requests
    ...
}
```

The lack of any time-based cooldown mechanism means that an account that is liquidatable can be processed repeatedly in rapid succession. During periods of high market volatility, temporary price fluctuations might push an account below the maintenance margin requirement. Without a cooldown, the account can be liquidated multiple times without giving the trader an opportunity to add margin or otherwise mitigate the situation.

Impact

- **Repeated Liquidations:** Traders may experience multiple liquidations in quick succession, which can drastically erode their collateral and lead to severe financial losses.
- **Unfair Penalties:** A lack of a grace period means that temporary market perturbations can trigger unnecessary liquidations, penalizing traders for momentary liquidity issues.
- **Market Manipulation:** Malicious actors could exploit this rapid-fire liquidation mechanism to manipulate the market or repeatedly target specific accounts.

Tools Used

- **Manual Code Review**
- **Static Analysis Tools**
- **Simulation Testing**.

Recommendations

- **Implement a Cooldown Period:** Introduce a time-based cooldown interval in the liquidation mechanism, during which an account cannot be re-liquidated. This can be achieved by storing the timestamp of the last liquidation attempt for each account.

  Example:

  ```
  require(
      block.timestamp >= tradingAccount.lastLiquidationTimestamp + COOLDOWN_PERIOD,
      "Liquidation cooldown period has not elapsed"
  );
  ```

- **Grace Period for Margin Top-ups:** Allow a grace period after an account becomes liquidatable to provide traders with the opportunity to add margin or adjust their positions.

- **Update State Variables:** Ensure that upon liquidation, the account's `lastLiquidationTimestamp` (or equivalent storage variable) is updated accordingly.
- **Extensive Testing:** Develop tests to verify that the cooldown mechanism effectively prevents repeated liquidations in rapid succession and works correctly under various market conditions.

M-14. Lack of Minimum Liquidation Amount Check in LiquidationBranch.sol:liquidateAccounts()

Summary

The liquidation process does not enforce a minimum liquidation amount, which can lead to the processing of extremely small (dust) positions. This inefficiency can result in disproportionate gas consumption relative to the liquidation size, and may also be exploited by malicious liquidators to force unnecessary liquidations on small positions.

Vulnerability Details

Within the `liquidateAccounts(...)` function, the protocol calls `tradingAccount.deductAccountMargin` as shown in the snippet:

```
ctx.liquidatedCollateralUsdX18 = tradingAccount.deductAccountMargin(
    TradingAccount.DeductAccountMarginParams({
        pnlUsdX18: ctx.accountTotalUnrealizedPnlUsdX18.abs().intoUD60x18().add(
            ctx.requiredMaintenanceMarginUsdX18
        ),
        // No minimum amount check
        ...
    })
);
```

In this implementation, there is no safeguard to prevent the liquidation of very small amounts. This means that even dust-sized positions can be liquidated, leading to inefficient gas usage. Moreover, a malicious liquidator might exploit this by repeatedly liquidating small positions to unnecessarily incur gas costs or to disrupt normal liquidation behavior for traders with minor positions.

Impact

- **Gas Inefficiency:** Liquidating small positions can lead to gas consumption that is not economically justified, making the process inefficient for liquidators.
- **Exploitation Risk:** Malicious actors might target dust positions for strategic reasons, possibly leading to a higher frequency of liquidations on trivial amounts.
- **User Impact:** Traders with small positions may face repeated, inefficient liquidations, potentially increasing transaction costs and overall losses.

Tools Used

- **Manual Code Review:** A detailed inspection of the liquidation flow in `LiquidationBranch.sol` uncovered the absence of a minimum liquidation amount check.
- **Static Analysis Tools:** Automated tools were used to trace fund flow and identify areas where minimum thresholds could enhance efficiency.
- **Simulation Testing:** Scenarios were simulated to analyze gas costs relative to very small liquidation amounts, confirming the potential inefficiency.

Recommendations

- **Introduce a Minimum Liquidation Threshold:** Implement a check that prevents the liquidation process from proceeding if the collateral or position size is below a pre-defined minimum value. For example:

```
require(
    collateralAmount >= MIN_LIQUIDATION_AMOUNT,
    "Liquidation amount below minimum threshold"
);
```

- **Dynamic Threshold Adjustment:** Consider implementing dynamic minimum thresholds based on market conditions and gas prices to maintain economic efficiency.
- **Document Behavior:** Clearly document the minimum liquidation requirements in protocol documentation and user interfaces, so that liquidators and traders are aware of the bounds.
- **Thorough Testing:** Develop and execute tests to ensure that the minimum threshold logic prevents dust liquidations without affecting legitimate liquidation events.

M-15. No Partial Liquidation Support in LiquidationBranch,sol::liquidateAccounts()

Summary

The liquidation mechanism currently mandates the complete closure of a position rather than supporting partial liquidation. This all-or-nothing approach may force traders to liquidate their entire market exposure unnecessarily, potentially resulting in higher losses than if only a fraction of the position were closed to restore adequate margin levels.

Vulnerability Details

The code forces full liquidation by computing the liquidation size as the negative of the entire open position:

```
// Save inverted sign of open position size to prepare for closing the position
ctx.liquidationSizeX18 = -ctx.oldPositionSizeX18;
```

Later, the entire position is cleared:

```
// ... later in the code ...
position.clear();
```

This indicates that the system does not support partial liquidation where only a portion of the position is closed to bring the account back in compliance with margin requirements. The absence of flexibility in liquidation size means that even when a small position reduction would suffice, the protocol liquidates the entire position, potentially exacerbating losses.

Impact

- **Excessive Losses for Traders:** Forcing a full liquidation may lead to higher-than-necessary losses, as traders lose the benefit of retaining some exposure which might still be profitable.
- **Inefficient Risk Management:** Traders lose the opportunity to adjust positions gradually and recover by partially closing positions, leading to more aggressive and costly liquidation outcomes.
- **Market Confidence Erosion:** The inability to execute partial liquidations may deter market participants who value flexibility in managing risk, potentially affecting overall market liquidity.

Tools Used

- **Manual Code Review**
- **Static Analysis Tools**
- **Simulation Testing**

Recommendations

- **Implement Partial Liquidation Logic:** Modify the liquidation mechanism to allow for closing only a portion of a position when appropriate. This may involve:
  - Calculating the minimum liquidation size necessary to restore the margin above the maintenance threshold.
  - Adjusting the position size by the calculated amount instead of clearing it entirely.
- **Dynamic Liquidation Parameters:** Introduce parameters that determine the optimal fraction of the position to liquidate based on the margin shortfall and current market conditions.
- **Update Event Logging:** Enhance event emissions to include details of the partial liquidation, such as the portion liquidated and the remaining position size.
- **Comprehensive Testing:** Develop unit and integration tests to cover various partial liquidation scenarios ensuring that the system behaves as expected under different market conditions.

Low Risk Findings

L-01. Possible Reentrancy During Fee Transfers in FeeDistributionBranch.sol::receiveMarketFee

Summary

The function receiveMarketFee(...) is vulnerable to reentrancy attacks due to external calls before state updates.

Vulnerability Details

- The function calls `transferFrom` before updating state, allowing potential reentrancy.
- A malicious token could re-enter contract functions before state updates are finalized.

```
IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
```

Impact

Double-counting or bypassing supply checks, leading to potential fund loss.

Tools Used

Manual code review.

Recommendations

- Implement reentrancy guards.
- Update state before making external calls.

L-02. Missing input validation for DEX swap strategy paths in FeeDistributionBranch.sol::performMultiDexSwap

Summary

The function _performMultiDexSwap lacks validation for the length of dexSwapStrategyIds, which could lead to errors.

Vulnerability Details

- No validation that dexSwapStrategyIds.length matches assets.length - 1.
- Could result in array out-of-bounds errors or incomplete swaps.

```
 function _performMultiDexSwap(
AssetSwapPath.Data memory swapPath,
uint256 assetAmount
)
internal
returns (uint256)
{
address[] memory assets = swapPath.assets;
uint128[] memory dexSwapStrategyIds = swapPath.dexSwapStrategyIds;
// No validation that dexSwapStrategyIds.length == assets.length - 1
```

```
uint256 amountIn = assetAmount;
for (uint256 i; i < assets.length - 1; i++) {
DexSwapStrategy.Data storage dexSwapStrategy =
DexSwapStrategy.loadExisting(dexSwapStrategyIds[i]);
...
}
return amountIn;
}
```

Impact

Potential protocol invariants breakage and incomplete swaps.

Tools Used

Manual code review.

Recommendations

- Validate the length of `dexSwapStrategyIds` against `assets.length - 1`.
- Ensure `assets` contains at least two elements.

L-03. In FeeDistributionBranch.sol::_handleWethRewardDistribution rounding error could lead to fund loss

Summary

Rounding errors in `_handleWethRewardDistribution` could result in trapped funds.

Vulnerability Details

- Rounding errors in fixed-point arithmetic could leave dust amounts in the contract.
- Over time, these errors could accumulate to significant amounts.

```
UD60x18 receivedProtocolWethRewardX18 =
receivedWethX18.mul(feeRecipientsSharesX18);
UD60x18 receivedVaultsWethRewardX18 =
receivedWethX18.mul(ud60x18(Constants.MAX_SHARES).sub(feeRecipientsSharesX18)
);
// calculate leftover reward
UD60x18 leftover =
receivedWethX18.sub(receivedProtocolWethRewardX18).sub(receivedVaultsWethRewa
rdX18);
```

Impact

Accumulation of unclaimable funds due to rounding errors.

Tools Used

Manual code review.

Recommendations

- Implement mechanisms to handle rounding errors.
- Consider using a more precise arithmetic library.

L-04. In FeeDistributionBranch.sol::claimFees() there is no reentrancy guard.

Summary

The function `claimFees(...)` lacks reentrancy protection, posing a risk if future features are introduced.

Vulnerability Details

- The function performs token transfers without reentrancy guards.
- Although current logic sets the user's fee balance to zero, future changes could introduce risks.

```
IERC20(weth).safeTransfer(msg.sender, amountToClaim);
```

Impact

Potential reentrancy attacks if new features or external calls are added.

Tools Used

Manual code review.

Recommendations

- Add reentrancy guards to critical functions.
- Review future changes for reentrancy risks.

L-05. In FeeDistributionBranch.sol::_performMultiDexSwap absence of leftover or partial handling in multi-dex swap can lock tokens

Summary

The function `_performMultiDexSwap` does not handle leftover tokens, potentially locking them.

Vulnerability Details

- The function does not account for leftover tokens after swaps.
- Accumulated rounding errors could result in discrepancies.

```
function _performMultiDexSwap(
AssetSwapPath.Data memory swapPath,
uint256 assetAmount
) internal returns (uint256) {
// ...calls executeSwapExactInputSingle in a loop...
// The function does not handle leftover or partial tokens, returning the
final amount only.
}
```

Impact

Locked tokens due to unhandled leftovers.

Tools Used

Manual code review.

Recommendations

- Implement handling for leftover tokens.
- Ensure all tokens are accounted for after swaps.

L-06. In FeeDistributionBranch.sol::getAssetValue there is missing Asset Price Validation

Summary

The function `getAssetValue` retrieves asset prices without validating their freshness or reliability.

Vulnerability Details

- The function does not check the validity of asset prices.
- Stale or manipulated prices could affect asset valuations.

```
function getAssetValue(address asset, uint256 amount) public view returns
(uint256 value) {
Collateral.Data storage collateral = Collateral.load(asset);
UD60x18 priceX18 = collateral.getPrice();
// ... calculations without price validation
}
```

Impact

Inaccurate asset valuations impacting fee distributions.

Tools Used

Manual code review.

Recommendations

- Validate the freshness and reliability of asset prices.
- Implement checks for price validity.

L-07. In FeeDistributionBranch.sol::_handleWethRewardDistribution there is precision Loss in Fee Distribution Calculations

Summary

The contract's fee distribution calculations involve multiple decimal conversions and mathematical operations, which can lead to precision loss, particularly in reward distribution.

Vulnerability Details

- The calculations for distributing WETH rewards involve converting and multiplying decimal values.
- Precision loss can occur due to rounding errors in fixed-point arithmetic operations.
- The contract attempts to handle leftover dust by adding it to vault rewards, but accumulated rounding errors over time could lead to discrepancies.

```
 UD60x18 receivedProtocolWethRewardX18 =receivedWethX18.mul(feeRecipientsSharesX18);
UD60x18 receivedVaultsWethRewardX18 =receivedWethX18.mul(ud60x18(Constants.MAX_SHAR
```

Impact

Minor discrepancies in fee distribution could accumulate over time, potentially leading to small amounts of unallocated or misallocated funds.

Tools Used

Manual code review and analysis of arithmetic operations.

Recommendations

- Use a more precise arithmetic library to minimize rounding errors.
- Implement a mechanism to periodically reconcile and correct any discrepancies in fee distribution.

- Consider logging or monitoring the accumulation of dust amounts to ensure they remain within acceptable limits.

L-08. Precision Loss in Fee Calculations in StabilityBranch.sol

Summary

The fee calculation logic uses multiplication followed by division with fixed-point arithmetic. Although supported by libraries like PRB Math, the order of operations can lead to rounding issues, particularly for very small amounts.

Vulnerability Details

The fee math is implemented as follows:

```
function getFees(assetsAmountOut, price) {
  // Convert base fee from USD to asset amount.
  let baseFee = baseFeeUsd / price;

  // Calculate dynamic swap fee with rounding up.
  let swapFee = divUp(assetsAmountOut * swapSettlementFeeBps, BPS_DENOMINATOR);

  return { baseFee, swapFee };
}
```

Impact

- **Rounding Errors:** In transactions with small amounts, rounding differences could lead to slight overcharging or undercharging.
- **Accumulated Inaccuracies:** While minor per transaction, repeated errors could have a cumulative impact over many swaps.

Tools Used

- Static analysis
- Mathematical verification and simulation
- Fuzz testing

Recommendations

- **Rounding Optimization:** Review the order of arithmetic operations to minimize precision loss.
- **Edge Case Testing:** Perform detailed tests on small volume swaps to understand and adjust for rounding.
- **Library Enhancements:** Consider using higher precision arithmetic or specialized libraries if necessary.

L-09. Refund Does Not Reimburse Gas Costs in StabilityBranch::refundSwap

Summary

When a swap request expires, the `refundSwap` function refunds the USD amount net of the base fee. Thus, users incur the cost of gas for both initiating and refunding the swap, in addition to losing the base fee even though no swap service was rendered.

Vulnerability Details

A simplified outline of the refund process:

```
function refundSwap(request) {
  let baseFee = tokenSwapData.baseFeeUsd;
  // Base fee is deducted from the refunded amount.
  let refundAmount = request.amountIn - baseFee;
  // The base fee is distributed to protocol fee recipients.
  distributeProtocolFee(baseFee);
  transfer(usdToken, user, refundAmount);
}
```

Impact

- **Double Gas Expense:** Users pay gas twice—once to initiate the swap and again to obtain a refund.
- **Loss of Funds:** The base fee is forfeited despite no service being provided, reducing overall user funds and satisfaction.

Tools Used

- Manual code review
- Transaction cost analysis
- Simulation tests

Recommendations

- **Refund Design Revision:** Consider refunding the base fee if a swap is not fulfilled.
- **User Incentives:** Explore compensatory mechanisms to cover gas costs incurred on refunds.
- **Clear Communication:** Ensure UI and documentation clearly explain the fee structure to users.

L-10. In StabilityBranch.sol there is lack of Re-entrancy Protection for External Token Transfers

Summary

The contract performs external token transfers (e.g., via `safeTransferFrom`) without a re-entrancy guard. If the token contract is nonstandard or maliciously designed, it could trigger a callback and re-enter the swap function,

potentially causing state inconsistencies.

Vulnerability Details

An abstraction of the vulnerable section:

```
function executeSwap() {
  // Critical state updates are performed.
  updateSwapState();

  // Then an external call is made without a re-entrancy guard.
  token.safeTransferFrom(vault.indexToken, address(this), amountOut + protocolRewar

  // Further processing continues...
}
```

Impact

- **Re-entrancy Attacks:** A malicious token contract could re-enter the function during the external call and manipulate the state or drain funds.
- **Unexpected Behavior:** The absence of guards may lead to double spending, faulty fee calculations, or collateral mismanagement.

Tools Used

- Manual review
- Re-entrancy analysis tools
- Static analysis

Recommendations

- **Implement Re-entrancy Guard:** Add a standard `nonReentrant` modifier (using OpenZeppelin's `ReentrancyGuard`) around functions performing external calls.
- **Order of Operations:** Ensure that all critical state changes are made before any external call and consider using a Checks-Effects-Interactions pattern.

L-11. No Partial Fill Support for Swap Requests in StabilityBranch.sol::fulfillSwap

Summary

Swap requests are designed to execute atomically; if the computed output is less than the user-defined `minAmountOut`, the whole swap is reverted. There is no mechanism for executing partial fills when liquidity is insufficient to meet the user's minimum requirement.

Vulnerability Details

The current logic is as follows:

```
function fulfillSwap(user, requestId, priceData, engine) {
  let amountOut = calculateAmountOut(priceData);
  let minAmountOut = getMinAmountOut(requestId);

  // If the computed output is less than expected, the swap reverts completely.
  if (amountOut < minAmountOut) {
      throw Error("SlippageCheckFailed");
  }

  // Otherwise, the swap is fully executed.
  executeSwap(amountOut);
}
```

Impact

- **User Experience Degradation:** Users experience abrupt failures even when a partial swap might be acceptable in volatile or low-liquidity scenarios.
- **Liquidity Inefficiency:** The inability to fill orders partially may lead to underutilization of available liquidity, reducing overall market efficiency.

Tools Used

- Manual review
- Functional testing with edge-case simulations
- Economic scenario modeling

Recommendations

- **Implement Partial Fills:** Enhance the swap logic to allow for proportional or partial execution of swap requests when liquidity is insufficient for full execution.
- **User-configurable Slippage:** Allow users to set parameters that accommodate partial fills or accept increased slippage for partial execution.
- **User Interface and Documentation:** Clearly communicate how partial fills operate and any trade-offs associated with them.

L-12. Missing zero-amount validation in fee calculations in StabilityBranch.sol::getFeesForAssetsAmountOut

Summary

The functions `getFeesForAssetsAmountOut` and `getFeesForUsdTokenAmountIn` perform fee calculations without checking that their input amounts are non-zero. While other parts of the system may implicitly catch zero-value scenarios, lacking explicit validation can lead to ambiguous or unexpected arithmetic outcomes, potentially causing issues in extreme edge cases.

Vulnerability Details

In the following function, there is no validation to ensure that the input amounts (or related parameters such as the price) are non-zero:

```
function getFeesForAssetsAmountOut(
UD60x18 assetsAmountOutX18,
UD60x18 priceX18
) public view returns (UD60x18 baseFeeX18, UD60x18 swapFeeX18) {
// load swap data
UsdTokenSwapConfig.Data storage tokenSwapData =
UsdTokenSwapConfig.load();
// convert the base fee in usd to the asset amount to be charged
baseFeeX18 = ud60x18(tokenSwapData.baseFeeUsd).div(priceX18);
// calculates the swap fee portion rounding up
swapFeeX18 = Math.divUp(
assetsAmountOutX18.mul(ud60x18(tokenSwapData.swapSettlementFeeBps)),
ud60x18Convert(Constants.BPS_DENOMINATOR));
}
```

Impact

- **Ambiguous Fee Calculations:** Zero-value inputs could lead to undefined or unintended arithmetic behavior, such as divisions by zero or affecting fee rounding.
- **Edge-Case Vulnerabilities:** Although not directly exploitable, missing validation opens the door for potential issues or future vulnerabilities if the fee calculation logic is altered.
- **Non-adherence to Best Practices:** Financial calculations should always validate critical inputs to ensure clarity and strict adherence to intended constraints.

Tools Used

- **Manual Code Review:** A thorough review of fee functions revealed the lack of input validations.
- **Static Analysis Tools:** Tools flagged missing input checks within arithmetic operations.
- **Fuzz Testing:** Fuzzing with zero-value inputs confirmed the need for explicit validations.

Recommendations

- **Implement Explicit Zero-Value Checks:** Update the functions to require that input amounts and prices are non-zero. For example:

```
require(assetsAmountOutX18 != UD60x18_ZERO, "Zero asset output not allowed");
require(priceX18 != UD60x18_ZERO, "Zero price not allowed");
```

- **Consistent Validation Across Functions:** Apply similar zero-value checks in `getFeesForUsdTokenAmountIn` and other related fee modules.
- **Update Documentation:** Clearly document that the input parameters must be non-zero, outlining the expected requirements for proper fee calculation.
- **Thorough Testing:** Add unit tests to ensure that the contract reverts when zero values are provided, thereby safeguarding against unexpected behavior.

L-13. Race Condition in LiquidationBranch.sol between checkLiquidatableAccounts() and liquidateAccounts()

Summary

The separation between the state-reading function (`checkLiquidatableAccounts`) and the state-altering liquidation function (`liquidateAccounts`) introduces a race condition. The list of liquidatable accounts obtained from `checkLiquidatableAccounts` can quickly become outdated if an account's collateral is modified before the actual liquidation call, leading to liquidators potentially incurring gas fees for accounts that are no longer eligible.

Vulnerability Details

The `checkLiquidatableAccounts` function scans a range of account IDs and returns those that appear to be below the maintenance margin at the time of the call:

```
function checkLiquidatableAccounts(
    uint256 lowerBound,
    uint256 upperBound
)
    external
    view
    returns (uint128[] memory liquidatableAccountsIds)
{
    // ... checks array of potential account IDs, returning those that appear below
}
```

However, between the time these account IDs are returned and when `liquidateAccounts` is executed, the on-chain state (e.g., user margin deposits or position modifications) may change. Consequently, by the time liquidation is attempted, some accounts might no longer meet the criteria. Although `liquidateAccounts` revalidates each account and skips non-liquidatable ones, liquidators still risk wasting transaction fees or experiencing operational confusion due to stale data.

Impact

- **Wasted Gas Fees:** Liquidators might incur unnecessary transaction costs by attempting to liquidate accounts that are no longer liquidatable.
- **Operational Inefficiency:** The discrepancy between the view call and the actual liquidation process may lead to confusion and reduce the reliability of automated liquidation mechanisms.
- **User Experience:** Liquidators relying on stale data may face challenges in timing their liquidation transactions, affecting overall market efficiency.

Tools Used

- **Manual Code Review:** Detailed analysis of the separation between state-reading and state-changing functions highlighted the potential race condition.
- **Static Analysis Tools:** Automated tools confirmed the non-atomic nature of the account selection and liquidation processes.
- **Simulation Testing:** Scenario-based simulations demonstrated how rapid state changes could lead to discrepancies between the data returned by `checkLiquidatableAccounts` and the actual conditions during `liquidateAccounts`.

Recommendations

- **Atomic Liquidation Mechanism:** Consider designing an atomic liquidation process where the selection and liquidation of accounts occur in a single transaction or under a single state snapshot.
- **Timestamp or Nonce Inclusions:** Include a timestamp or version number with the liquidation data from `checkLiquidatableAccounts` to help liquidators assess the freshness of the data.
- **Enhanced Documentation:** Clearly document the inherent race condition, advising liquidators to account for possible state changes and the associated risks.
- **Off-Chain Coordination:** Utilize off-chain monitoring and validation services that can provide more synchronized liquidation triggers, reducing the dependency on stale on-chain data.

L-14. Potential Storage Read Optimization in Loop in LiquidationBranch.sol::liquidateAccounts()

Summary

The liquidation process in `liquidateAccounts` repeatedly reads certain storage variables inside the loop for each account processed. This can result in higher gas consumption when multiple accounts are processed in a single transaction, indicating an area where gas optimization improvements can be made.

Vulnerability Details

Within the loop inside the `liquidateAccounts` function, repeated storage reads occur as part of the liquidation eligibility check. For instance, the liquidation fee stored in the protocol configuration is accessed each time an account is processed:

```
if (
    !TradingAccount.isLiquidatable(
        ctx.requiredMaintenanceMarginUsdX18,
        ctx.marginBalanceUsdX18,
        ctx.liquidationFeeUsdX18
    )
) {
    continue;
}
```

Since `ctx.liquidationFeeUsdX18` (derived from the protocol configuration) is a constant value during the transaction, fetching this value repeatedly from storage increases the overall gas cost. This inefficiency is particularly relevant when processing a large number of accounts during a liquidation event.

Impact

- **Increased Gas Costs:** Repeated storage reads result in higher gas consumption per transaction, reducing the economic efficiency of bulk liquidations.
- **Scalability Concerns:** As the number of accounts processed increases, the cumulative gas cost rise may become significant, impacting the protocol's performance during high-stress liquidation events.
- **Operational Efficiency:** Extra gas costs may deter liquidators from processing large batches, potentially delaying necessary liquidations.

Tools Used

- **Manual Code Review:** A focused review of the `liquidateAccounts` function identified repeated storage reads in critical loops.
- **Gas Profiling Tools:** Tools such as Hardhat and Truffle's gas reporter helped quantify the gas cost implications of repeated storage access.
- **Static Analysis:** Automated analysis tools confirmed that certain state variables are read repeatedly within tight loops, suggesting an area for optimization.

Recommendations

- **Cache Immutable Values:** Load values that do not change throughout the transaction (e.g., the liquidation fee and configuration parameters) into local memory variables prior to entering the loop. For example:

```
UD60x18 cachedLiquidationFeeUsdX18 = ctx.liquidationFeeUsdX18;
// use cachedLiquidationFeeUsdX18 in liquidatable check instead of repeatedly rea
```

- **Optimize Loop Structure:** Review the loop for any additional storage reads that can be minimized by caching values in memory.
- **Benchmark Gas Usage:** Perform bench tests before and after optimization to assess the reduction in gas consumption, and adjust the logic accordingly.

- **Document Changes:** Update documentation to reflect any changes in the gas optimization strategy, ensuring future developers are aware of the cost-saving measures employed.

L-15. Precision Loss for Margin Balance Calculations in LiquidationBranch.sol::liquidateAccounts()

Summary

The contract performs multiple fixed-point arithmetic operations using high-precision libraries (`UD60x18` and `SD59x18`) to compute margin balances and liquidation thresholds. However, repeated operations may accumulate rounding errors, potentially leading to small inaccuracies in margin calculations. These imprecisions could result in accounts being liquidated slightly above or below the intended threshold.

Vulnerability Details

During the liquidation process, the contract calculates the notional value of positions as follows:

```
ctx.accountPositionsNotionalValueX18[j] = ctx.oldPositionSizeX18.abs().intoUD60x18
```

Despite using fixed-point libraries designed for high precision, each conversion and multiplication might introduce minute rounding differences. When these operations stack across multiple accounts or complex calculations, the resulting deviation, although small, could impact the accuracy of margin balance computations. This may inadvertently cause an account to be deemed liquidatable (or vice versa) due to accumulated precision errors.

Impact

- **Margin Misjudgment:** Minor rounding discrepancies could lead to accounts being liquidated even when they are marginally solvent, or being spared liquidation when they should be flagged.
- **User Dissatisfaction:** Traders may experience unexpected liquidations or delays, eroding trust in the protocol's fairness.
- **System Reliability:** Over time and across many transactions, small inaccuracies could aggregate, potentially affecting overall system stability and risk assessments.

Tools Used

- **Manual Code Review:** Detailed inspection of the arithmetic operations and their use of `UD60x18` and `SD59x18` libraries.
- **Static Analysis Tools**
- **Simulation Testing**

Recommendations

- **Review and Standardize Rounding Behavior:** Ensure that all arithmetic operations involving fixed-point numbers implement a consistent rounding strategy (e.g., always rounding up or down) to mitigate the accumulation of errors.

- **Precision Audit:** Conduct a thorough audit of all margin-related calculations to quantify the maximum potential deviation caused by rounding errors and confirm that it remains within acceptable bounds.
- **Enhanced Testing:** Introduce detailed unit tests and simulations specifically targeting the precision of margin calculations, ensuring that any discrepancies remain negligible.
- **Documentation:** Clearly document any known limitations of the fixed-point arithmetic used within the protocol to manage expectations regarding precision and its potential impact on liquidation decisions.