Polynomial Protocol contest Findings & Analysis Report

2023-08-01

Table of contents

- [H-07] Missing totalFunds update in LiquidityPool's `OpenShort()`, causing LiquidityPool token holder to lose a portion of their token value
- [H-08] Incorrect calculation of `usedFunds` in LiquidityPool leads to lower than expected token price
- [H-09] Excessive trading fees can result in 99.9% collateral loss for the trader
- [H-10] Function `hedgePositions` is incorrect, because it missed the `queuedPerpSize` in the calculation
- [H-11] KangarooVault `QueuedWithdraw` Denial of Service
- [H-12] Denial of service of Liquiditypool `QueuedWithdrawals`
- [H-13] Exchange: `totalFunding` calculation should be done with a simple multiplication operation instead of a `wadMul` operation
- [H-14] Inexpedient liquidatable logic that could have half liquidable position turns fully liquidable instantly

- Medium Risk Findings (19)

  - [M-01] Exchange Rate can be manipulated if positions are big enough for a long enough time
  - [M-02] Users can receive less collateral than expected from liquidations
  - [M-03] `KangarooVault.initiateDeposit`, `KangarooVault.processDepositQueue`, `KangarooVault.initiateWithdrawal`, and `KangarooVault.processWithdrawalQueue` functions do not use `whenNotPaused` modifier
  - [M-04] `LiquidityPool._getDelta` and `LiquidityPool._calculateMargin` functions should execute `require(!isInvalid && spotPrice > 0)` instead of `require(!isInvalid || spotPrice > 0)`
  - [M-05] Some functions that call `Exchange.getMarkPrice` function do not check if `Exchange.getMarkPrice` function's returned `markPrice` is 0

- [M-06] `KangarooVault._resetTrade` , `LiquidityPool.rebalanceMargin` , and `LiquidityPool._getTotalMargin` functions should check `isInvalid` , which is returned by external perp market contract's `remainingMargin` function, and revert if such `isInvalid` is true
- [M-07] `usedFunds` can be greater than `totalFunds` in contract `KangarooVault` , which leads to `KangarooVault` being unable to close its trades and users being unable to withdraw
- [M-08] LiquidityPool can be DoS when a complete withdrawal is performed
- [M-09] Short positions with minimum collateral can be liquidated even though `canLiquidate()` returns `false`
- [M-10] Malicious users can exploit deposit and withdrawal queueing in `KangarooVault` and `LiquidityPool` contracts to force exorbitant transaction fees
- [M-11] The Liquidity Pool will lack margin of Synthetix perpetual market, if `liquidationBufferRatio` of contract `PerpsV2MarketSettings` from Synthetix is updated
- [M-12] Attacker can post-running attack to prevent `LiquidityPool` from hedging by orders of PerpMarket
- [M-13] Attacker can transfer all SUSD's balance of LiquidityPool as margin to Synthetix perpetual market, and break the actions of users until the pool is rebalanced by the authority
- [M-14] Possible spamming attack in opening or closing Long or Short Positions in `Exchange.openTrade`
- [M-15] First Depositors will incurr a rebalancing Loss
- [M-16] Supply drain of PowerPerp tokens through liquidations
- [M-17] Users' collateral could get stuck permanently after fully closing short trades
- [M-18] Lack of price validity check from Synthetix results in loss of funds while liquidating
- [M-19] Collateral removal not possible

- Low Risk and Non-Critical Issues

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Polynomial Protocol smart contract system written in Solidity. The audit took place between March 13—March 20 2023.

Wardens

36 Wardens contributed reports to the Polynomial Protocol audit:

1. 0x52
2. 0xRobocop
3. 0xSmartContract
4. 0xbepresent
5. Bauer
6. CRYP70
7. DadeKuma
8. Diana
9. GalloDaSballo
10. Josiah
11. KIntern_NA (TrungOre and duc)
12. Lirios
13. MiloTruck
14. Nyx
15. PaludoX0
16. RaymondFam
17. Rolezn
18. Sathish9098
19. __141345__
20. adriro

21. auditor0517
22. bin2chen
23. btk
24. bytes032
25. carlitox477
26. chaduke
27. csanuragjain
28. joestakey
29. juancito
30. kaden
31. peakbolt
32. peanuts
33. rbserver
34. sakshamguruji
35. sorrynotsorry

This audit was judged by Dravee.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 33 unique vulnerabilities. Of these vulnerabilities, 14 received a risk rating in the category of HIGH severity and 19 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 14 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Polynomial Protocol audit repository, and is composed of 12 smart contracts written in the Solidity programming language and includes 1849 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

High Risk Findings (14)

[H-01] `Exchange._liquidate` function can cause liquidator to burn too much `powerPerp` tokens

Submitted by [rbserver]()

When calling the following `Exchange._liquidate` function, `uint256 totalCollateralReturned = shortCollateral.liquidate(positionId, debtRepaying, msg.sender)` is executed.

```
    function _liquidate(uint256 positionId, uint256 debtRepaying
        uint256 maxDebtRepayment = shortCollateral.maxLiquidatab
        require(maxDebtRepayment > 0);
        if (debtRepaying > maxDebtRepayment) debtRepaying = maxI
```

```
        IShortToken.ShortPosition memory position = shortToken.s

        uint256 totalCollateralReturned = shortCollateral.liquic

        address user = shortToken.ownerOf(positionId);

        uint256 finalPosition = position.shortAmount - debtRepay
        uint256 finalCollateralAmount = position.collateralAmour

        shortToken.adjustPosition(positionId, user, position.col

        pool.liquidate(debtRepaying);
        powerPerp.burn(msg.sender, debtRepaying);
        ...
    }
```

In the following `ShortCollateral.liquidate` function, when executing `uint256 collateralClaim = debt.mulDivDown(markPrice, collateralPrice)`, where `debt` is `debtRepaying`, `collateralClaim` can be high if `collateralPrice` has become much lower comparing to `markPrice`, such as due to a market sell-off that causes the collateral to be worth much less than before. In this case, `totalCollateralReturned` can be high as well, which can cause `totalCollateralReturned > userCollateral.amount` to be true. When such condition is true, `totalCollateralReturned = userCollateral.amount` is executed, and only `userCollateral.amount` is transferred to the liquidator after executing `ERC20(userCollateral.collateral).safeTransfer(user, totalCollateralReturned)`.

```
    function liquidate(uint256 positionId, uint256 debt, address
        external
        override
        onlyExchange
        nonReentrant
        returns (uint256 totalCollateralReturned)
    {
        UserCollateral storage userCollateral = userCollaterals[

        bytes32 currencyKey = synthetixAdapter.getCurrencyKey(us
        Collateral memory coll = collaterals[currencyKey];

        (uint256 markPrice,) = exchange.getMarkPrice();
```

```
        (uint256 collateralPrice,) = synthetixAdapter.getAssetPr
        uint256 collateralClaim = debt.mulDivDown(markPrice, col
        uint256 liqBonus = collateralClaim.mulWadDown(coll.liqBo
        totalCollateralReturned = liqBonus + collateralClaim;
        if (totalCollateralReturned > userCollateral.amount) tot
        userCollateral.amount -= totalCollateralReturned;

        ERC20(userCollateral.collateral).safeTransfer(user, tota
        ...
    }
```

Back in the `Exchange._liquidate` function, the liquidator burns `debtRepaying` `powerPerp` tokens after `powerPerp.burn(msg.sender, debtRepaying)` is executed. However, in this situation, the liquidator only receives `userCollateral.amount` collateral tokens that are less than the collateral token amount that should be equivalent to `debtRepaying` `powerPerp` tokens but this liquidator still burns `debtRepaying` `powerPerp` tokens. As a result, this liquidator loses the extra `powerPerp` tokens, which are burnt, that are equivalent to the difference between `debtRepaying` `powerPerp` tokens' equivalent collateral token amount and `userCollateral.amount` collateral tokens.

Proof of Concept

The following steps can occur for the described scenario.

1. Alice calls the `Exchange._liquidate` function with `debtRepaying` being 1000e18.
2. When the `ShortCollateral.liquidate` function is called, `totalCollateralReturned > userCollateral.amount` is true, and `userCollateral.amount` collateral tokens that are equivalent to 500e18 `powerPerp` tokens are transferred to Alice.
3. When `powerPerp.burn(msg.sender, debtRepaying)` is executed in the `Exchange._liquidate` function, Alice burns 1000e18 `powerPerp` tokens.
4. Because Alice only receives `userCollateral.amount` collateral tokens that are equivalent to 500e18 `powerPerp` tokens, she loses 500e18 `powerPerp` tokens.

Tools Used

VSCode

Recommended Mitigation Steps

The `Exchange._liquidate` function can be updated to burn the number of `powerPerp` tokens that are equivalent to the actual collateral token amount received by the liquidator instead of burning `debtRepaying` `powerPerp` tokens.

mubaris (Polynomial) confirmed

[H-02] Hedging during liquidation is incorrect

Submitted by KIntern_NA

Hedging will not work as expected, and LiquidityPool will lose funds without expectation.

Proof of concept

1. When a short position is liquidated in contract Exchange, function `_liquidate` will be triggered. It will burn the power perp tokens and reduce the short position amount accordingly.

```
function _liquidate(uint256 positionId, uint256 debtRepaying) ir
    ...
    uint256 finalPosition = position.shortAmount - debtRepaying;
    uint256 finalCollateralAmount = position.collateralAmount -

    shortToken.adjustPosition(positionId, user, position.collate

    pool.liquidate(debtRepaying);
    powerPerp.burn(msg.sender, debtRepaying);
    ...
```

2. As you can see, it will decrease the size of short position by `debtRepaying`, and burn `debtRepaying` power perp tokens. Because of the same amount, the skew of `LiquidityPool` will not change.

3. However, `pool.liquidate` will be called, and `LiquidityPool` will be hedged with `debtRepaying` amount.

```
function liquidate(uint256 amount) external override onlyExchang
    (uint256 markPrice, bool isInvalid) = getMarkPrice();
    require(!isInvalid);

    uint256 hedgingFees = _hedge(int256(amount), true);
    usedFunds += int256(hedgingFees);

    emit Liquidate(markPrice, amount);
}
```

4. Therefore, LiquidityPool will be hedged more than it needs, and the position of `LiquidityPool` in the Perp Market will be incorrect (compared with what it should be for hedging).

Recommended Mitigation Steps

Should not hedge the LiquidityPool during liquidation.

[mubaris (Polynomial) confirmed]

[H-03] Short positions can be burned while holding collateral

Submitted by [MiloTruck], *also found by* [bin2chen], [chaduke], [0x52], [Bauer], *and* [0xRobocop]

Users can permanently lose a portion of their collateral due to a malicious attacker or their own mistake.

Vulnerability Details

In the `ShortToken` contract, `adjustPosition()` is used to handle changes to a short position's short or collateral amounts. The function also handles the burning of positions with the following logic:

```
position.collateralAmount = collateralAmount;
```

```
    position.shortAmount = shortAmount;

    if (position.shortAmount == 0) {
        _burn(positionId);
    }
```

Where:

- `collateralAmount` - New amount of collateral in a position.
- `shortAmount` - New short amount of a position.
- `positionId` - ERC721 `ShortToken` of a short position.

As seen from above, if a position's `shortAmount` is set to 0, it will be burned. Furthermore, as the code does not ensure `collateralAmount` is not 0 before burning, it is possible to burn a position while it still has collateral.

If this occurs, the position's owner will lose all remaining collateral in the position. This remaining amount will forever be stuck in the position as its corresponding `ShortToken` no longer has an owner.

Proof of Concept

In the `Exchange` contract, users can reduce a position's `shortAmount` using `closeTrade()` (`Exchange.sol#L100-L109`) and `liquidate()` (`Exchange.sol#L140-L148`). With these two functions, there are three realistic scenarios where a position with collateral could be burned.

1. User reduces his position's `shortAmount` to 0

A user might call `closeTrade()` on a short position with the following parameters:

- `params.amount` - Set to the position's short amount.
- `params.collateralAmount` - Set to any amount less than the position's total collateral amount.

This would reduce his position's `shortAmount` to 0 without withdrawing all of its collateral, causing him to lose the remaining amount.

Although this could be considered a user mistake, such a scenario could occur if a user does not want to hold a short position temporarily without fully withdrawing his collateral.

2. Attacker fully liquidates a short position

In certain situations, it is possible for a short position to have collateral remaining after a full liquidation (example in the coded PoC below). This collateral will be lost as full liquidations reduces a position's `shortAmount` to 0, thereby burning the position.

3. Attacker frontruns a user's `closeTrade()` transaction with a liquidation

Consider the following scenario:

- Alice has an unhealthy short position that is under the liquidation ratio and can be fully liquidated.
- To bring her position back above the liquidation ratio, Alice decides to partially reduce its short amount. She calls `closeTrade()` on her position with the following parameters:
    - `params.amount` - Set to 40% of the position's short amount.
    - `params.collateralAmount` - Set to 0.
- A malicious attacker, Bob, sees her `closeTrade()` transaction in the mempool.
- Bob frontruns the transaction, calling `liquidate()` with the following parameters:
    - `positionId` - ID of Alice's position.
    - `debtRepaying` - Set to 60% of Alice's position's short amount.
- Bob's `liquidate()` transaction executes first, reducing the short amount of Alice's position to 40% of the original amount.
- Alice's `closeTrade()` transaction executes, reducing her position's short amount by 40% of the original amount, thus its `shortAmount` becomes 0.

In the scenario above, Alice loses the remaining collateral in her short position as it is burned after `closeTrade()` executes.

Note that this attack is possible as long as an attacker can liquidate the position's remaining short amount. For example, if Alice calls `closeTrade()` with 70% of the position's short amount, Bob only has to liquidate 30% of its short amount.

Coded PoC

The code below contains three tests that demonstrates the scenarios above:

1. `testCloseBurnsCollateral()`
2. `testLiquidateBurnsCollateral()`
3. `testAttackerFrontrunLiquidateBurnsCollateral()`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

import {TestSystem, Exchange, ShortToken, ShortCollateral, MockE

contract PositionWithCollateralBurned is TestSystem {
    // Protocol contracts
    Exchange private exchange;
    ShortToken private shortToken;
    ShortCollateral private shortCollateral;

    // sUSD token contract
    MockERC20Fail private SUSD;

    // Intial base asset price
    uint256 private constant initialBaseAssetPrice = 1e18;

    // Users
    address private USER = user_1;
    address private ATTACKER = user_2;

    function setUp() public {
        // Deploy contracts
        deployTestSystem();
        initPool();
        initExchange();
```

```
        preparePool();

        exchange = getExchange();
        shortToken = getShortToken();
        shortCollateral = getShortCollateral();
        SUSD = getSUSD();

        // Set initial price for base asset
        setAssetPrice(initialBaseAssetPrice);

        // Mint sUSD for USER
        SUSD.mint(USER, 1e20);

        // Mint powerPerp for ATTACKER
        vm.prank(address(exchange));
        getPowerPerp().mint(ATTACKER, 1e20);
    }

    function testCloseBurnsCollateral() public {
        // Open short position
        uint256 shortAmount = 1e18;
        uint256 collateralAmount = 1e15;
        uint256 positionId = openShort(shortAmount, collateralAm

        // Fully close position without withdrawing any collater
        closeShort(positionId, shortAmount, 0, USER);

        // positionId still holds 1e15 sUSD as collateral
        (,, uint256 remainingCollateralAmount, ) = shortToken.sh
        assertEq(remainingCollateralAmount, collateralAmount);

        // positionId is already burned (ie. ownerOf reverts wit
        vm.expectRevert("NOT_MINTED");
        shortToken.ownerOf(positionId);
    }

    function testLiquidateBurnsCollateral() public {
        // USER opens short position with amount = 1e18, collate
        uint256 shortAmount = 1e18;
        uint256 positionId = openShort(1e18, 1e15, USER);

        // Base asset price rises by 35%
        setAssetPrice(initialBaseAssetPrice * 135 / 100);

        // USER's entire short position is liquidatable
        assertEq(shortCollateral.maxLiquidatableDebt(positionId)
```

```
        // ATTACKER liquidates USER's entire short position
        vm.prank(ATTACKER);
        exchange.liquidate(positionId, shortAmount);

        // positionId has no remaining debt, but still holds son
        (, uint256 remainingAmount, uint256 remainingCollateralA
        assertEq(remainingAmount, 0);
        assertGt(remainingCollateralAmount, 0);

        // positionId is already burned (ie. ownerOf reverts wit
        vm.expectRevert("NOT_MINTED");
        shortToken.ownerOf(positionId);
    }

    function testAttackerFrontrunLiquidateBurnsCollateral() publ
        // USER opens short position with amount = 1e18, collate
        uint256 shortAmount = 1e18;
        uint256 positionId = openShort(1e18, 1e15, USER);

        // Base asset price rises by 40%
        setAssetPrice(initialBaseAssetPrice * 140 / 100);

        // USER's short position is liquidatable
        assertEq(shortCollateral.maxLiquidatableDebt(positionId)

        // ATTACKER frontruns USER's closeTrade() transaction, l
        vm.prank(ATTACKER);
        exchange.liquidate(positionId, shortAmount * 60 / 100);

        // USER's closeTrade() transaction executes, reducing sh
        closeShort(positionId, shortAmount * 40 / 100, 0, USER);

        // positionId has no remaining debt, but still holds son
        (, uint256 remainingAmount, uint256 remainingCollateralA
        assertEq(remainingAmount, 0);
        assertGt(remainingCollateralAmount, 0);

        // positionId is already burned (ie. ownerOf reverts wit
        vm.expectRevert("NOT_MINTED");
        shortToken.ownerOf(positionId);
    }

    function openShort(
        uint256 amount,
        uint256 collateralAmount,
```

```
        address user
    ) internal returns (uint256 positionId) {
        Exchange.TradeParams memory tradeParams;
        tradeParams.amount = amount;
        tradeParams.collateral = address(SUSD);
        tradeParams.collateralAmount = collateralAmount;

        vm.startPrank(user);
        SUSD.approve(address(exchange), collateralAmount);
        (positionId, ) = exchange.openTrade(tradeParams);
        vm.stopPrank();
    }

    function closeShort(
        uint256 positionId,
        uint256 amount,
        uint256 collateralAmount,
        address user
    ) internal {
        Exchange.TradeParams memory tradeParams;
        tradeParams.amount = amount;
        tradeParams.collateral = address(SUSD);
        tradeParams.collateralAmount = collateralAmount;
        tradeParams.maxCost = 100e18;
        tradeParams.positionId = positionId;

        vm.startPrank(user);
        SUSD.approve(address(getPool()), tradeParams.maxCost);
        exchange.closeTrade(tradeParams);
        vm.stopPrank();
    }
}
```

Recommended Mitigation

Ensure that positions cannot be burned if they have any collateral:

```
-            if (position.shortAmount == 0) {
+            if (position.shortAmount == 0 && position.collatera
                 _burn(positionId);
             }
```

[mubaris (Polynomial) confirmed via duplicate issue #65]()

[Dravee (judge) commented]():

> The 3rd scenario isn't likely due to frontrunning not being an issue on Optimism. This report still brings the most value and is the most well presented.

[H-04] `KangarooVault.removeCollateral` updates storage without actually removing collateral, resulting in lost collateral

Submitted by joestakey, *also found by* juancito, auditor0517, bin2chen, KIntern_NA, chaduke, Bauer, *and* 0x52

The admin can call `KangarooVault.addCollateral` to add additional collateral to a Power Perp position.

```
File: src/KangarooVault.sol
424:     function addCollateral(uint256 additionalCollateral) ex
425:         SUSD.safeApprove(address(EXCHANGE), additionalColla
426:         EXCHANGE.addCollateral(positionData.positionId, adc
427:
428:         usedFunds += additionalCollateral;
429:         positionData.totalCollateral += additionalCollatera
430:
431:         emit AddCollateral(positionData.positionId, additic
432:     }
```

This transfers `SUSD` to the `EXCHANGE` and updates the `usedFunds` and `positionData.totalCollateral`

The function `KangarooVault.removeCollateral` allows the admin to remove collateral if a position is healthy enough.

```
File: src/KangarooVault.sol
436:     function removeCollateral(uint256 collateralToRemove) e
437:         (uint256 markPrice,) = LIQUIDITY_POOL.getMarkPrice(
438:     uint256 minColl = positionData.shortAmount.mulWadDown(m
439:         minColl = minColl.mulWadDown(collRatio);
```

```

            require(positionData.totalCollateral >= minColl + c

            usedFunds -= collateralToRemove;
            positionData.totalCollateral -= collateralToRemove;

            emit RemoveCollateral(positionData.positionId, coll
        }
```

The issue is that this function does not call `EXCHANGE.removeCollateral`. While it updates storage, it does not actually retrieve any collateral.

Impact

2 problems arising:

- `processWithdrawalQueue` will revert unexpectedly, as `usedFunds` will be lower than it should, leading to `availableFunds` being greater than the real balance of `SUSD` available (`KangarooVault.sol#L279`).
- the main problem: the amount of collateral “removed” in `removeCollateral` will be lost :

When closing a position in `KangarooVault._closePosition`, the amount of collateral to retrieve is written in `tradeParams.collateralAmount`. As you can see below, it is capped by `positionData.totalCollateral`, which was decremented in `removeCollateral`.

```
File: src/KangarooVault.sol
            if (amt >= positionData.shortAmount) {
                longPositionToClose = positionData.longPerp;

                tradeParams.amount = positionData.shortAmount;
                tradeParams.collateralAmount = positionData.tot
            } else {
                longPositionToClose = amt.mulDivDown(positionDa
                uint256 collateralToRemove = amt.mulDivDown(pos

                tradeParams.amount = amt;
                tradeParams.collateralAmount = collateralToRemc
            }
```

```

            SUSD.safeApprove(address(LIQUIDITY_POOL), maxCost);
            uint256 totalCost = EXCHANGE.closeTrade(tradeParams
```

This is the amount of collateral that will be transferred back to the trader (here the `KangarooVault`)

```
src/Exchange.sol _closeTrade()
shortCollateral.sendCollateral(params.positionId, params.cc
```

```
File: src/ShortCollateral.sol
        function sendCollateral(uint256 positionId, uint256 amc
            UserCollateral storage userCollateral = userCollate

            userCollateral.amount -= amount;

            address user = shortToken.ownerOf(positionId);

            ERC20(userCollateral.collateral).safeTransfer(user,
```

In conclusion, calling `removeCollateral` will result in that amount being lost.

Proof of Concept

Amend this test to `KangarooVault.t.sol`, which shows how collateral is not transferred upon calling `removeCollateral()`.

```
        function testCollateralManagement() public {
            uint256 amt = 1e18;
            uint256 collDelta = 1000e18;

            kangaroo.openPosition(amt, 0);
            skip(100);
            kangaroo.executePerpOrders(emptyData);
            kangaroo.clearPendingOpenOrders(0);

            (,,,,,,, uint256 initialColl,) = kangaroo.positionI
+            uint256 balanceBefore = susd.balanceOf(address(kar

```

```
        kangaroo.addCollateral(collDelta);
+        uint256 balanceAfter = susd.balanceOf(address(kang
+        assertEq(collDelta, balanceBefore - balanceAfter);
        (,,,,,,, uint256 finalColl,) = kangaroo.positionDat

        assertEq(finalColl, initialColl + collDelta);

+        uint256 balanceBefore2 = susd.balanceOf(address(ka
        kangaroo.removeCollateral(collDelta);
+        uint256 balanceAfter2 = susd.balanceOf(address(kan
+        assertEq(0, balanceAfter2 - balanceBefore2); //@au

        (,,,,,,, uint256 newColl,) = kangaroo.positionData(

        assertEq(newColl, initialColl);
    }
```

Tools Used

Manual Analysis, Foundry

Recommended Mitigation

Ensure `Exchange.removeCollateral` is called:

```
File: src/KangarooVault.sol
    function removeCollateral(uint256 collateralToRemove) e
        (uint256 markPrice,) = LIQUIDITY_POOL.getMarkPrice(
        uint256 minColl = positionData.shortAmount.mulWadDo
        minColl = minColl.mulWadDown(collRatio);

        require(positionData.totalCollateral >= minColl + c

        usedFunds -= collateralToRemove;
        positionData.totalCollateral -= collateralToRemove;

+        EXCHANGE.removeCollateral(positionData.positionId,
        emit RemoveCollateral(positionData.positionId, coll
    }
```

[mubaris (Polynomial) confirmed via duplicate issue #111]

[Dravee (judge) increased severity to High]

[H-05] Uneven deduction of performance fee causes some KangarooVault users to lose part of their token value

Submitted by [peakbolt]

In `KangarooVault._resetTrade()`, a `performanceFee` is charged upon closing of all positions, on the `premiumCollected`. This is inconsistent with `getTokenPrice()` as `premiumCollected` is factored in the token price computation, while the `performanceFee` is not. This leads to an uneven distribution of the `performanceFee` for the `KangarooVault` users.

Impact

That means a user can evade the `performanceFee` and steal some of the funds from the rest by triggering `processWithdraw()` before the `performanceFee` is deducted from `KangarooVault`. The remaining users will be shortchanged and lose part of their token value as they bear the charges from the performance fee.

Detailed Explanation

When all positions in `KangarooVault` are closed, `_resetTrade()` is triggered, which will proceed to deduct a `performanceFee` from the `premiumCollected`.

```
    function _resetTrade() internal {
        positionData.positionId = 0;
        (uint256 totalMargin,) = PERP_MARKET.remainingMargin(address
        PERP_MARKET.transferMargin(-int256(totalMargin));
        usedFunds -= totalMargin;

        uint256 fees = positionData.premiumCollected.mulWadDown(perf
        if (fees > 0) SUSD.safeTransfer(feeReceipient, fees);

        totalFunds += positionData.premiumCollected - fees;
        totalFunds -= usedFunds;

        positionData.premiumCollected = 0;
```

```
        positionData.totalMargin = 0;
        usedFunds = 0;
    }
```

KangarooVault.sol#L788-L789

However, only `premiumCollected` is factored in the `getTokenPrice()` computation but not the `performanceFee`. That means the premiums are distributed among the users via token price, while the performance fee is not.

```
    function getTokenPrice() public view returns (uint256) {
        if (totalFunds == 0) {
            return 1e18;
        }

        uint256 totalSupply = getTotalSupply();
        if (positionData.positionId == 0) {
            return totalFunds.divWadDown(totalSupply);
        }

        uint256 totalMargin;

        (uint256 markPrice, bool isInvalid) = EXCHANGE.getMarkPrice(
        require(!isInvalid);
        (totalMargin, isInvalid) = PERP_MARKET.remainingMargin(addre
        require(!isInvalid);

        uint256 totalValue = totalFunds + positionData.premiumCollec
        totalValue -= (usedFunds + markPrice.mulWadDown(positionData

        return totalValue.divWadDown(totalSupply);
    }
```

KangarooVault.sol#L358-L359

Proof of Concept

Add the following imports and test case to `test/KangarooVault.t.sol`

```
    import {IVaultToken} from "../src/interfaces/IVaultToken.sol";
```

```
function testKangarooPerformanceFee() public {
    uint256 amt = 231e18;
    IVaultToken vaultToken = IVaultToken(kangaroo.VAULT_TOKEN())

    // deposit equal value for both user_2 and user 3 into Kanga
    uint256 depositAmt = 10e18;
    susd.mint(user_2, depositAmt);
    vm.startPrank(user_2);
    susd.approve(address(kangaroo), depositAmt);
    kangaroo.initiateDeposit(user_2, depositAmt);
    assertEq((vaultToken.balanceOf(user_2) * kangaroo.getTokenPr
    vm.stopPrank();

    susd.mint(user_3, depositAmt);
    vm.startPrank(user_3);
    susd.approve(address(kangaroo), depositAmt);
    kangaroo.initiateDeposit(user_3, depositAmt);
    assertEq((vaultToken.balanceOf(user_2) * kangaroo.getTokenPr
    vm.stopPrank();

    skip(14500);
    kangaroo.processDepositQueue(2);


    // Open position at KangarooVault and execute the orders
    kangaroo.openPosition(amt, 0);
    skip(100);
    kangaroo.executePerpOrders(emptyData);
    kangaroo.clearPendingOpenOrders(0);

    // Simulate price drop to trigger profit from premium collec
    setAssetPrice(initialPrice - 100e18);

    // initiate withdrawal for both user_2 and user_3
    vm.prank(user_2);
    kangaroo.initiateWithdrawal(user_2, depositAmt);

    vm.prank(user_3);
    kangaroo.initiateWithdrawal(user_3, depositAmt);

    skip(14500);

    // close all position with gain from premium collection
    kangaroo.closePosition(amt, 1000000e18);
    skip(100);
```

```
        kangaroo.executePerpOrders(emptyData);


        // user_2 frontrun clearPendingCloseOrders() to withdraw at
        kangaroo.processWithdrawalQueue(1);
        assertEq(vaultToken.balanceOf(user_2), 0);
        assertEq(susd.balanceOf(user_2), 969382134314627414l);

        // This will trigger resetTrade and deduct performance Fee
        kangaroo.clearPendingCloseOrders(0);

        // user_3's withdrawal was processed but at a lower token pr
        kangaroo.processWithdrawalQueue(1);
        assertEq(vaultToken.balanceOf(user_3), 0);
        assertEq(susd.balanceOf(user_3),965576808821137284l);

        // This shows that user_3 was shortchanged and lost part of
        // despite starting with equal token balance
        assertGt(susd.balanceOf(user_2), susd.balanceOf(user_3));
    }
```

Recommended Mitigation Steps

Consider changing the following in `KangarooVault.sol#L359`

```
        totalValue -= (usedFunds + markPrice.mulWadDown(positionData
```

to

```
        totalValue -= (usedFunds + markPrice.mulWadDown(positionData
```

[mubaris (Polynomial) confirmed]

[H-06] Division by zero error causes KangarooVault to be DoS with funds locked inside

Submitted by [peakbolt]

`KangarooVault` can be DoS with funds locked in the contract due to a division by zero error in `getTokenPrice()` as it does not handle the scenario where `getTotalSupply()` is zero.

Impact

Funds will be locked within the `KangarooVault` (as shown in the PoC below) and it is not able to recover from the DoS.

That is because, to recover from the DoS, it requires increasing total supply through minting of new tokens via deposits. However, that is not possible as `initiateDeposit()` relies on `getTokenPrice()`.

Also, we cannot withdraw the remaining funds as there are no more `VaultTokens` left to burn.

Detailed Explanation

`getTokenPrice()` will attempt to perform a division by `getTotalSupply()` when `totalFunds != 0` and `positionId == 0`. This scenario is possible when there are remaining funds in `KangarooVault` when all positions are closed and all vault token holders withdrawn their funds.

```
    function getTokenPrice() public view returns (uint256) {
        if (totalFunds == 0) {
            return 1e18;
        }

        uint256 totalSupply = getTotalSupply();
        if (positionData.positionId == 0) {
            return totalFunds.divWadDown(totalSupply);
        }
```

Proof of Concept

Add the following imports and test case to `test/Kangaroo.Vault.t.sol`

```
    function testKangarooDivisionByZero() public {
```

```
        uint256 amt = 231e18;

        // Open position to decrease availableFunds for withdrawals.
        kangaroo.openPosition(amt, 0);
        skip(100);
        kangaroo.executePerpOrders(emptyData);
        kangaroo.clearPendingOpenOrders(0);

        // initiate user withdrawal
        // this will be a partial withdrawal due to the open positic
        vm.prank(user_1);
        kangaroo.initiateWithdrawal(user_1, 5e23);
        kangaroo.processWithdrawalQueue(1);

        // close all position
        kangaroo.closePosition(amt, 1000000e18);
        skip(100);
        kangaroo.executePerpOrders(emptyData);
        kangaroo.clearPendingCloseOrders(0);

        // Complete remaining withdrawals of funds.
        // this will reduce totalSupply to zero and later cause a di
        kangaroo.processWithdrawalQueue(1);

        /// prepare for new deposit
        vm.startPrank(user_1);
        susd.approve(address(kangaroo), 5e23);

        // This deposit will revert due to division by zero.
        vm.expectRevert();
        kangaroo.initiateDeposit(user_1, 5e23);
        vm.stopPrank();

        // KangarooVault is now DoS and some funds are locked in it
        assertEq(susd.balanceOf(address(kangaroo)), 168969);
    }
```

Recommended Mitigation Steps

Fix `getTokenPrice()` to handle the scenario when `totalSupply()` is zero.

[Dravee (judge) commented](#):

> Feels extremely similar to https://github.com/code-423n4/2023-03-polynomial-findings/issues/157 by the same warden, but the impact is on a different contract and requires a different POC.
> Won't flag as a duplicate for now.

mubaris (Polynomial) confirmed

Dravee (judge) commented:

> Will keep as high due to the warden showing in this case a direct impact on assets.

[H-07] Missing totalFunds update in LiquidityPool's `OpenShort()`, causing LiquidityPool token holder to lose a portion of their token value

Submitted by peakbolt, also found by auditor0517, 0xbepresent, kaden, and 0xRobocop

The function `openShort()` in `LiquidityPool.sol` is missing an update to `totalFunds`, to increase `LiquidityPool` funds by the collected net fees.

Impact

As a result of the missing increment to `totalFunds`, the `availableFunds` in the `LiquidityPool` will be lower. This will impact the token price, causing a lower token price on `openShort()` trades. This will result in `LiquidityPool` token holders to lose part of their token value.

Detailed Explanation

The function `openShort()` is supposed to increase the `totalFunds` by `(feesCollected - externalFee)` as the trading fees is paid by the trader, via a deduction of the `tradeCost`.

```
        totalCost = tradeCost - fees;
```

```
    SUSD.safeTransfer(user, totalCost);
```

Proof of Concept

Add the following imports and test case to `test/LiquidityPool.Trades.t.sol`

```
import {wadMul} from "solmate/utils/SignedWadMath.sol";
import {IPerpsV2Market} from "../src/interfaces/synthetix/IPerps

function testLiquidityPoolOpenShort() public {
    uint256 amount = 1e18;

    (uint256 markPrice, bool isInvalid) = pool.getMarkPrice();
    uint256 tradeCost = amount.mulWadDown(markPrice);
    uint256 fees = pool.orderFee(int256(amount));
    uint256 delta = pool.getDelta();
    int256 hedgingSize = wadMul(int256(amount), int256(delta));
    IPerpsV2Market perp = pool.perpMarket();
    (uint256 hedgingFees, ) = perp.orderFee(hedgingSize, IPerpsV
    uint256 feesCollected = fees - hedgingFees;
    uint256 externalFee = feesCollected.mulWadDown(pool.devFee()

    uint256 totalFundsBefore = pool.totalFunds();
    int256 usedFundsBefore = pool.usedFunds();

    // Open a Short trade
    openShort(amount, amount * 1000, user_1);

    // Calculated expected totalFunds and usedFunds
    uint256 expectedTotalFunds = totalFundsBefore + feesCollecte
    uint256 marginRequired = tradeCost + hedgingFees;
    int256 expectedUsedFunds = usedFundsBefore + int256(tradeCos

    // This is incorrect as LiquidityPool's totalFunds is suppos
    assertLt(pool.totalFunds(), expectedTotalFunds);

    // LiquidityPool's UsedFunds is also wrong and is higher tha
    assertGt(pool.usedFunds(), expectedUsedFunds);

    uint256 poolAvailableFunds = pool.totalFunds() - uint256(poo
    uint256 expectedAvailableFunds = expectedTotalFunds - uint25
```

```
        // LiquidityPool's available funds is wrong and is less thar
        assertLt(poolAvailableFunds, expectedAvailableFunds);
        assertEq(poolAvailableFunds, expectedAvailableFunds - hedgir


        // LiquidityPool's available funds is wrong and is also less
        assertLt(poolAvailableFunds, susd.balanceOf(address(pool)));

        // LiquidityPool's available fund is expected to be the same
        assertEq(expectedAvailableFunds, susd.balanceOf(address(pool

        // LiquidityPool Token price is less than expected.
        assertLt(pool.getTokenPrice(), getExpectedTokenPrice(expecte

    }

    function getExpectedTokenPrice(uint256 expectedTotalFunds, int25
        (uint256 markPrice,) = pool.getMarkPrice();
        uint256 totalValue = expectedTotalFunds;
        uint256 totalSupply = lqToken.totalSupply() + pool.totalQueu
        uint256 amountOwed = markPrice.mulWadDown(powerPerp.totalSup
        uint256 amountToCollect = markPrice.mulWadDown(shortToken.tc
        //uint256 totalMargin = _getTotalMargin();

        (uint256 totalMargin,) = perp.remainingMargin(address(pool))

        totalValue += totalMargin + amountToCollect;
        totalValue -= uint256((int256(amountOwed) + expectedUsedFunc

        expectedTokenPrice = totalValue.divWadDown(totalSupply);
    }
```

Recommended Mitigation Steps

Add the following to update `totalFunds` with the net fee collection.

```
        totalFunds += feesCollected - externalFee;
```

rivalq (Polynomial) disputed and commented:

> Fee part is included in usedFunds, At any time pool's total funds is not just included in `totalFunds`, but some part of it is in `usedFunds`, note that

> usedFunds can be negative too.

[Dravee (judge) commented](#):

> As this issue was raised by several wardens, I'm willing to give this the benefit of the doubt and would like to ask the sponsor @rivalq to view it a second time. Perhaps looking through duplicated issues? They are mostly low quality and badly explained but might be on to something. https://github.com/code-423n4/2023-03-polynomial-findings/issues/46 is the duplicate with the most arguments.

> I'd like to use this issue to bring attention to another issue, https://github.com/code-423n4/2023-03-polynomial-findings/issues/117, as it actually says the opposite (that `openShort` is done right but `closeLong` has an extra update that shouldn't exist). So, this one, which I repeat isn't a duplicate but actually an opposite finding, might be right.

[mubaris (Polynomial) confirmed and commented](#):

> I'm confirming this from our side, although I think it's a Medium risk similar to `#117`.

[Dravee (judge) commented](#):

> As this is still considered a loss of funds for users, I'll keep it as High.

[H-08] Incorrect calculation of `usedFunds` in LiquidityPool leads to lower than expected token price

Submitted by [peakbolt](#), also found by [auditor0517](#) and [KIntern_NA](#)

In `LiquidityPool.sol`, the functions `openLong()`, `closeLong()`, `openShort()` and `closeShort()` do not deduct `hedgingFees` from `usedFunds` to offset the `hedgingFees` that was added due to `_hedge()`.

Impact

The missing deduction of `hedgingFees` will increase the `usedFunds` in `LiquidityPool`, thus reducing the `availableFunds`. This leads to a lower token price, causing `LiquidityPool` token holders to be shortchanged, losing a portion of their token value.

Detailed Explanation

Passive users can provide liquidity to the Liquidity Pool to earn exchange fees, while traders can take long or short position against the Exchange.

The trade functions `openLong()` , `closeLong()` , `openShort()` and `closeShort()` in `LiquidityPool.sol` will call `_hedge` to hedge the liquidity pool's exposure during a trade.

The `hedge()` will actually increase `usedFunds` by the margin required, which includes `hedgingFees` as these are transfered over to the `perpMarket` for the hedging.

```
uint256 marginRequired = _calculateMargin(hedgingSize) + hec
usedFunds += int256(marginRequired);
require(usedFunds <= 0 || totalFunds >= uint256(usedFunds));

perpMarket.transferMargin(int256(marginRequired));
```

Using the `OpenLong()` trade as an example, the trader transfers the `tradeCost + fees` to `LiquidityPool` , which includes the premium, `hedgingFees` , `feesCollected` and `externalFee` . That means the `hedgingFees` that was transfered in `_hedge()` is actually provided by the trader and not the `LiquidityPool` .

Hence, the `usedFunds` should be reduced by `hedgingFees` to offset the addition in `_hedge()` .

```
uint256 fees = orderFee(int256(amount));
totalCost = tradeCost + fees;

SUSD.safeTransferFrom(user, address(this), totalCost);

uint256 hedgingFees = _hedge(int256(amount), false);
uint256 feesCollected = fees - hedgingFees;
```

```
    uint256 externalFee = feesCollected.mulWadDown(devFee);

    SUSD.safeTransfer(feeReceipient, externalFee);

    usedFunds -= int256(tradeCost);
    totalFunds += feesCollected - externalFee;
```

Proof of Concept

Then add the following imports and test case to

`test/LiquidityPool.Trades.t.sol`

```
import {wadMul} from "solmate/utils/SignedWadMath.sol";
import {IPerpsV2Market} from "../src/interfaces/synthetix/IPerps


function testLiquidityPoolFundCalculation() public {
    uint256 longAmount = 1e18;

    (uint256 markPrice, bool isInvalid) = pool.getMarkPrice();
    uint256 tradeCost = longAmount.mulWadDown(markPrice);
    uint256 fees = pool.orderFee(int256(longAmount));
    uint256 delta = pool.getDelta();
    int256 hedgingSize = wadMul(int256(longAmount), int256(delta
    IPerpsV2Market perp = pool.perpMarket();
    (uint256 hedgingFees, ) = perp.orderFee(hedgingSize, IPerps\
    uint256 feesCollected = fees - hedgingFees;
    uint256 externalFee = feesCollected.mulWadDown(pool.devFee()

    uint256 totalFundsBefore = pool.totalFunds();
    int256 usedFundsBefore = pool.usedFunds();

    // Open a Long trade
    openLong(longAmount, longAmount * 1000, user_1);

    // Calculated expected totalFunds and usedFunds
    uint256 expectedTotalFunds = totalFundsBefore + feesCollecte
    uint256 marginRequired = tradeCost + hedgingFees;
    int256 expectedUsedFunds = usedFundsBefore - int256(tradeCos

    // This is correct as the pool will increase by net fee (fee
    assertEq(pool.totalFunds(), expectedTotalFunds);
```

```
        // LiquidityPool's UsedFunds is wrong and is higher than exp
        assertGt(pool.usedFunds(), expectedUsedFunds);

        uint256 poolAvailableFunds = pool.totalFunds() - uint256(poo
        uint256 expectedAvailableFunds = expectedTotalFunds - uint25

        // LiquidityPool's available funds is wrong and is less thar
        assertLt(poolAvailableFunds, expectedAvailableFunds);
        assertEq(poolAvailableFunds, expectedAvailableFunds - hedgir

        // LiquidityPool's available funds is wrong and is also less
        assertLt(poolAvailableFunds, susd.balanceOf(address(pool)));

        // LiquidityPool's available fund is expected to be the same
        assertEq(expectedAvailableFunds, susd.balanceOf(address(pool

        // LiquidityPool Token price is less than expected.
        assertLt(pool.getTokenPrice(), getExpectedTokenPrice(expecte

    }

    function getExpectedTokenPrice(uint256 expectedTotalFunds, int25
        (uint256 markPrice,) = pool.getMarkPrice();
        uint256 totalValue = expectedTotalFunds;
        uint256 totalSupply = lqToken.totalSupply() + pool.totalQueu
        uint256 amountOwed = markPrice.mulWadDown(powerPerp.totalSup
        uint256 amountToCollect = markPrice.mulWadDown(shortToken.to
        //uint256 totalMargin = _getTotalMargin();

        (uint256 totalMargin,) = perp.remainingMargin(address(pool))

        totalValue += totalMargin + amountToCollect;
        totalValue -= uint256((int256(amountOwed) + expectedUsedFunc

        expectedTokenPrice = totalValue.divWadDown(totalSupply);
    }
```

Recommended Mitigation Steps

Deduct `hedgingFees` from `usedFunds` to offset the `hedgingFees` added in `_hedge()`.

For example, in openLong change

```
        usedFunds -= int256(tradeCost);
```

to

```
        usedFunds -= int256(tradeCost) - int256(hedgingFees);
```

mubaris (Polynomial) confirmed

[H-09] Excessive trading fees can result in 99.9% collateral loss for the trader

Submitted by bytes032

This issue can lead to traders losing all their collateral due to excessive fees when opening and closing larger trades.

Proof of Concept

When the exchange opens/closes trades, it defines the order fees to be paid by the user using the `orderFee` function. The function takes into account both the hedging fee, which covers the cost of managing the risk associated with holding the position, and the trading fee, which accounts for the costs incurred when executing the trade.

The function works as follows:

1. It first calculates the `delta`, a factor representing the rate at which the position size changes.
2. Then, it computes the `futuresSizeDelta` by multiplying the input `sizeDelta` with the calculated `delta`.
3. It calls the `perpMarket.orderFee` function to get the `hedgingFee` and a boolean flag, `isInvalid`, which indicates if the trade is invalid.
4. It checks if the trade is valid by ensuring `isInvalid` is not set to `true`. If the trade is invalid, it will throw an exception.

5. It retrieves the `markPrice` from the `exchange.getMarkPrice` function, which represents the current market price of the asset.
6. The function calculates the `valueExchanged` by multiplying the `markPrice` with the absolute value of `sizeDelta`. This represents the total value of the trade.
7. It computes the `tradeFee` using the `getSlippageFee` function, which calculates the fee based on the size of the trade.
8. The `tradingFee` is calculated by multiplying the `tradeFee` with `valueExchanged`.
9. Finally, the function returns the sum of the `hedgingFee` and `tradingFee` as the total fee for the trade.

By combining both the hedging fee and trading fee, the `orderFee` function comprehensively calculates the costs associated for the trade.

The problem here lies in the `getSlippageFee` (`LiquidityPool.sol#L367-L374`) function. The purpose of the `getSlippageFee` function is to calculate the slippage fee for a trade based on the change in position size, represented by the input `sizeDelta`. The slippage fee is a cost associated with executing a trade that accounts for the potential price impact and liquidity changes due to the size of the order. It helps ensure that the trading platform can effectively manage the impact of large trades on the market.

```
function getSlippageFee(int256 sizeDelta) public view returr
    // ceil(s/100) * baseFee

    uint256 size = sizeDelta.abs();
    uint256 region = size / standardSize;

    if (size % standardSize != 0) region += 1;
    return region * baseTradingFee;
}
```

The function works as follows:

1. It first calculates the absolute value of `sizeDelta` to ensure the size is a positive value, regardless of whether the trade is a buy or sell.
2. It then divides the `size` by `standardSize` (set by default to (`LiquidityPool.sol#L142`) 1e20), to determine the number of regions the trade occupies. This constant value represents the size of a “standard” trade, which is used to categorize trades into different regions based on their size.
3. If there is a remainder after dividing `size` by the constant value, it increments the `region` by 1. This ensures that even partial regions are taken into account when calculating the fee.
4. Finally, the function calculates the slippage fee by multiplying the `region` with `baseTradingFee` (currently set (`TestSystem.sol#L202`) to 6e15 in the test suite). This constant value represents the base fee for each region.

By calculating the slippage fee based on the size of the trade, the `getSlippageFee` function helps account for the potential impact of the trade on the market. The larger the trade, the more regions it occupies, resulting in a higher slippage fee. This approach incentivizes traders to be mindful of the size of their trades and the potential impact they may have on market liquidity and pricing.

The issue is that region can get really big (depending on the trade) so that the openTrade/closeTrades `orderFee` could >= 100% meaning all the collateral of the user will be used to pay taxes.

While I completely understand the intent of higher taxes for bigger trades, I think there should be a limit where the trade won’t get opened if a certain threshold is passed.

I’ve built a PoC with Foundry using the protocol test suite with a few comments here and there to represent the following cases:

```
function depositToPool(address user, uint256 sum) internal {
    susd.mint(user, sum);
    vm.startPrank(user);
    susd.approve(address(pool), sum);
    pool.deposit(sum, user);
    vm.stopPrank();
}
```

```
function openShort(uint256 amount, uint256 collateral, addre
    internal
    returns (uint256 positionId, Exchange.TradeParams memory
{
    tradeParams.amount = amount;
    tradeParams.collateral = address(susd);
    tradeParams.collateralAmount = collateral;
    tradeParams.minCost = 0;

    vm.startPrank(user);
    susd.approve(address(exchange), collateral);
    (positionId,) = exchange.openTrade(tradeParams);
    vm.stopPrank();
}

 function testSimpleShortCloseTrade() public {
    depositToPool(user_2, 1000e18 * 25000);
    uint256 pricingConstant = exchange.PRICING_CONSTANT();
    uint256 expectedPrice = initialPrice.mulDivDown(initialP
    uint256 multiplier = 165;
    uint256 collateralAmount = (expectedPrice * 2) * multipl
    uint256 shortAmount = multiplier * 1e18;
    susd.mint(user_1, collateralAmount);
    console.log("CollateralAmount", collateralAmount / 1e18)
    console.log("ShortAmount", shortAmount / 1e18);
    console.log("User_1 sUSD balance", susd.balanceOf(user_1


    console.log("*** OPEN TAX ***");
    (uint256 positionId,) = openShort(shortAmount, collatera
    console.log("*** OPEN TAX ***\n");

    console.log("*** CLOSE TAX ***");
    closeShort(positionId, shortAmount, type(uint256).max, c
    console.log("*** CLOSE TAX ***");

    console.log("User_1 sUSD balance", susd.balanceOf(user_1
}
```

I'm going to use the formula `(orderFee * 2) / collateralAmount` to calculate the tax percentage. For each case, I'll just modify the `multiplier` variable.

Case 1: (multiplier 165)
CollateralAmount 237600

ShortAmount 165
The tax % is: 1,7996633

Running the tests, yields the following results.

```
CollateralAmount 237600
ShortAmount 165
User_1 sUSD balance 237600
*** OPEN TAX ***
==orderFee to be paid by the user== 2138
*** OPEN TAX ***

*** CLOSE TAX ***
==orderFee to be paid by the user== 2138
*** CLOSE TAX ***
User_1 sUSD balance 233323
```

Case 2: (multiplier 1650)
CollateralAmount 2376000
ShortAmount 1650
The tax % is: 10,8

Running the tests yields the following results.

```
CollateralAmount 2376000
ShortAmount 1650
User_1 sUSD balance 2376000
*** OPEN TAX ***
==orderFee to be paid by the user== 128304
*** OPEN TAX ***

*** CLOSE TAX ***
==orderFee to be paid by the user== 128304
*** CLOSE TAX ***
User_1 sUSD balance 2119392
```

Case 3: (multiplier 16500)
CollateralAmount 23760000
ShortAmount 16500
The tax % is: 99,6

```
CollateralAmount 23760000
ShortAmount 16500
User_1 sUSD balance 23760000
*** OPEN TAX ***
==orderFee to be paid by the user== 11832480
*** OPEN TAX ***

*** CLOSE TAX ***
==orderFee to be paid by the user== 11832480
*** CLOSE TAX ***
User_1 sUSD balance 95040
```

Case 3 proves the point that a user can potentially lose all his collateral just by paying taxes when opening/closing a trade.

Tools Used

Manual review, Foundry

Recommended Mitigation Steps

This invariant is heavily influenced by the calculation in `LiquidityPool.sol#L371`. In case 1 the region is 1, in case 2 is 16 and in case 3 is 165.

To address this issue, a protocol-wide maximum amount of taxes to be paid in a trade (total of open + close) should be established. If the calculated fee exceeds this threshold, the transaction should be reverted instead of opening the trade for the user. This will help protect traders from losing all their collateral due to excessive trading fees.

[mubaris (Polynomial) confirmed]

[H-10] Function `hedgePositions` is incorrect, because it missed the `queuedPerpSize` in the calculation

Submitted by [KIntern_NA]

Function `hedgePositions` is incorrect, leads to the hedging will not work as expected, and `LiquidityPool` can lose funds without expectation.

Proof of concept

Let's see function `hedgePositions` in `LiquidtyPool` contract:

```
function hedgePositions() external override requiresAuth nonReer
    int256 currentPosition = _getTotalPerpPosition();
    int256 skew = _getSkew();
    uint256 delta = _getDelta();
    int256 requiredPosition = wadMul(skew, int256(delta));

    int256 newPosition = requiredPosition - currentPosition;
    int256 marginDelta = int256(_calculateMargin(newPosition));

    if (requiredPosition.abs() < currentPosition.abs()) {
        marginDelta = -marginDelta;
    }

    usedFunds += marginDelta;

    perpMarket.transferMargin(marginDelta);
    _placeDelayedOrder(newPosition, false);

    emit HedgePositions(currentPosition, requiredPosition, margi
}
```

`currentPosition` is the sum of: the current position size of `LiquidityPool` in Synthetix and the delta size of the current delayed order which was submitted into Synthetix perp market.

```
function _getTotalPerpPosition() internal view returns (int2
    IPerpsV2MarketBaseTypes.Position memory position = perpM
    IPerpsV2MarketBaseTypes.DelayedOrder memory delayedOrder

    positionSize = position.size + delayedOrder.sizeDelta;
}
```

However, `currentPosition` missed the variable `queuedPerpSize`, is the total amount of pending size delta (waiting to be submitted).

Then `_placeDelayedOrder` will be called with the wrong `newPosition`, leads to the position size of pool can get a large deviation. The hedging will not be safe anymore.

Scenario:

- `_getTotalPerpPosition` = 0, `requiredPosition` = 1000, `queuedPerpSize` = 1000
- `newPosition` is calculated incorrectly to be 1000 (since it missed `queuedPerpSize`)
- It calls `_placeDelayedOrder(1000, false)`, then `queuedPerpSize` increase to be 2000
- After executing all delayed orders, position size of LiquidityPool = 2000 (incorrect hedging)
- `newPosition` should be -1000 in this case

Recommended Mitigation Steps

`currentPosition` should be `_getTotalPerpPosition()` + `queuedPerpSize` in function `hedgePositions`.

[mubaris (Polynomial) confirmed](#)

[H-11] KangarooVault `QueuedWithdraw` Denial of Service

Submitted by Lirios

When the `KangarooVault` has an open position, any withdrawals that are initiated, are queued.

`QueuedWithdrawals` work in two steps.

1. A user initialtes the Withdrawal via `initiateWithdrawal` (`KangarooVault.sol#L215`). This burns the `VaultToken` and `if (positionData.positionId != 0)` (`KangarooVault.sol#L225`) adds the request to the `withdrawalQueue`.
2. `processWithdrawalQueue()` can be called to process requests in the `withdrawalQueue` that have passed `minWithdrawDelay` to transfer the SUSD tokens to the user.

If the processing of a `QueuedWithdraw` entry in the `withdrawalQueue` reverts, the `queuedWithdrawalHead` (`KangarooVault.sol#L331`) will never increase and further processing of the queue will be impossible. This means that any users that have placed a QueuedWithdraw after the reverting entry will have lost their vaultToken without receiving their SUSD.

Proof of Concept

When calling the `initiateWithdrawal()` function, the user can provide an address of the receiver of funds.

When processing the withdrawal queue, the contracts does all the required checks, and then transfers the SUSD (`KangarooVault.sol#L322`) to the provided user.

If we look at the Synthetix sUSD token and it's target implementation we will find that the SUSD token transfer code is:

sUSD MultiCollateralSynth:L723-L739

```
function _internalTransfer(
    address from,
    address to,
    uint value
```

```
) internal returns (bool) {
    /* Disallow transfers to irretrievable-addresses. */
    require(to != address(0) && to != address(this) && to != adc

    // Insufficient balance will be handled by the safe subtract
    tokenState.setBalanceOf(from, tokenState.balanceOf(from).suk
    tokenState.setBalanceOf(to, tokenState.balanceOf(to).add(val

    // Emit a standard ERC20 transfer event
    emitTransfer(from, to, value);

    return true;
}
```

This means any SUSD transfer to the SUSD proxy or implementation contract, will result in a revert.

An attacker can use this to make a `initiateWithdrawal()` request with `user=sUSDproxy` or `user=sUSD_MultiCollateralSynth`. Any user that request a Withdrawal via `initiateWithdrawal()` after this, will lose their vault tokens without receiving their SUSD.

The attacker can do this at any time, or by frontrunning a specific (large) `initiateWithdrawal()` request.

To test it, a check is added to the mock contract that is used for SUSD in the test scripts:

```
diff --git a/src/test-helpers/MockERC20Fail.sol b/src/test-helpe
index e987f04..1ce10ec 100644
--- a/src/test-helpers/MockERC20Fail.sol
+++ b/src/test-helpers/MockERC20Fail.sol
@@ -18,6 +18,9 @@ contract MockERC20Fail is MockERC20 {
     }

     function transfer(address receiver, uint256 amount) public
+
+        require(receiver != address(0xDfA2d3a0d32F870D87f8A0d7A
+
         if (forceFail) {
             return false;
```

```
}
```

In the `KangarooVault.t.sol` test script, the following test was added to demonstrated the issue:

```
// add to top of file:
import {IVaultToken} from "../../src/interfaces/IVaultToken.sol'

// add to KangarooTest Contract:
    function testWithdrawalDOS() public {

        IVaultToken vault_token = kangaroo.VAULT_TOKEN();
        // make deposit for user_2
        susd.mint(user_2, 2e23);
        vm.startPrank(user_2);
        susd.approve(address(kangaroo), 2e23);
        kangaroo.initiateDeposit(user_2, 2e23);
        assertEq(vault_token.balanceOf(user_2),2e23);
        vm.stopPrank();


        // have vault open a position to force queued wthdrawals
        testOpen();

        // vault has  position opened, withdrawal will be queuec
        vm.startPrank(user_2);
        kangaroo.initiateWithdrawal(user_2, 1e23);
        assertEq(susd.balanceOf(user_2),0);
        assertEq(vault_token.balanceOf(user_2),1e23);

        // process withdrawalqueue, withdrawam should pass
        skip(kangaroo.minWithdrawDelay());
        kangaroo.processWithdrawalQueue(3);
        uint256 user_2_balance = susd.balanceOf(user_2);
        assertGt(user_2_balance,0);
        vm.stopPrank();


        // user 3 frontruns with fake/reverting withdrawal reque
        // to 0xDfA2d3a0d32F870D87f8A0d7AA6b9CdEB7bc5AdB (= SUSI
        // This will cause SUSD transfer to revert.
        vm.startPrank(user_3);
```

```
        kangaroo.initiateWithdrawal(0xDfA2d3a0d32F870D87f8A0d7AA
        vm.stopPrank();

        // user_2 adds another withdrawal request, after the att
        vm.startPrank(user_2);
        kangaroo.initiateWithdrawal(user_2, 1e23);
        assertEq(vault_token.balanceOf(user_2),0);


        // processWithdrawalQueue now reverts and no funds recei
        skip(kangaroo.minWithdrawDelay());
        vm.expectRevert(bytes("TRANSFER_FAILED"));
        kangaroo.processWithdrawalQueue(3);
        assertEq(susd.balanceOf(user_2),user_2_balance);
        assertEq(vault_token.balanceOf(user_2),0);
        vm.stopPrank();

    }
```

Tools Used

Manual review, forge

Recommended Mitigation Steps

The processing of `withdrawalQueue` should have a mechanism to handle reverting `QueuedWithdraw` entries. Either by skipping them and/or moving them to another `failedWithdrawals` queue.

[Dravee (judge) commented](#):

> Similar but different from https://github.com/code-423n4/2023-03-polynomial-findings/issues/103

> Somehow the import should be `import {IVaultToken} from "../src/interfaces/IVaultToken.sol";` (one step less), but the POC runs correctly after that.

[mubaris (Polynomial) confirmed](#)

[H-12] Denial of service of Liquiditypool `QueuedWithdrawals`

Submitted by Lirios, also found by bin2chen

The preferred way for withdrawals of the Liquiditypool is to do this via a withdrawal queue. According to Polynomial:

> Queuing will be the default deposit/withdraw mechanism (In the UI) and not planning to charge any fees for this mechanism
> Instant deposit / withdraw is mechanism is meant for external integrations in case if they don't want to track status of the queued deposit or withdraw

It is also stimulated to use `queueWithdraw()` over `withdraw()` by charging a withdrawalFee for direct withdrawals.

QueuedWithdrawals work in two steps.

1. A user initialtes the Withdrawal via `queueWithdraw()`. This burns the `liquidityTokens` and adds the request to the `withdrawalQueue`.
2. `processWithdraws()` can be called to process requests in the `withdrawalQueue` that have passed `minWithdrawDelay` to transfer the SUSD tokens to the user.

If the processing of a `QueuedWithdraw` in the `withdrawalQueue` reverts, the `queuedWithdrawalHead` (`LiquidityPool.sol#LL331C13-L331C33`) will never increase and further processing of the queue will be impossible. This means that any users that have placed a QueuedWithdraw after the reverting entry will have lost their liquiditytokens without receiving their SUSD.

Proof of Concept

When calling the `queueWithdraw()` function, the user can provide an address of the receiver of funds.

When processing the withdrawal queue, the contracts does all the required checks, and then transfers the SUSD (`LiquidityPool.sol#L311`) to the provided user.

If we look at the Synthetix sUSD token and it's target implementation we will find that the SUSD token transfer code is:

sUSD MultiCollateralSynth:L723-L739

```
    function _internalTransfer(
        address from,
        address to,
        uint value
    ) internal returns (bool) {
        /* Disallow transfers to irretrievable-addresses. */
        require(to != address(0) && to != address(this) && to !=

        // Insufficient balance will be handled by the safe subt
        tokenState.setBalanceOf(from, tokenState.balanceOf(from)
        tokenState.setBalanceOf(to, tokenState.balanceOf(to).adc

        // Emit a standard ERC20 transfer event
        emitTransfer(from, to, value);

        return true;
    }
```

This means any transfer to the SUSD proxy or implementation contract, will result in a revert.

An attacker can use this to make `queueWithdraw()` request with `user=sUSDproxy` or `user=sUSD_MultiCollateralSynth`. Any user that request a Withdrawal via `queueWithdraw()` after this, will lose their liquidity tokens without receiving their SUSD.

The attacker can do this at any time, or by frontrunning a specific (large) `queueWithdraw()` request.

To test it, a check is added to the mock contract that is used for SUSD in the test scripts to simulate the SUSD contract behaviour:

```
diff --git a/src/test-helpers/MockERC20Fail.sol b/src/test-helpe
index e987f04..1ce10ec 100644
```

```
--- a/src/test-helpers/MockERC20Fail.sol
+++ b/src/test-helpers/MockERC20Fail.sol
@@ -18,6 +18,9 @@ contract MockERC20Fail is MockERC20 {
     }

     function transfer(address receiver, uint256 amount) public
+
+        require(receiver != address(0xDfA2d3a0d32F870D87f8A0d7A
+
         if (forceFail) {
             return false;
         }
```

In the test/LiquidityPool.Deposits.t.sol test file, the following was added.
This results in a revert of the processWithdraws function and failing the test

```
iff --git a/test/LiquidityPool.Deposits.t.sol b/test/LiquidityPo
index 0bb6f5f..8d70c60 100644
--- a/test/LiquidityPool.Deposits.t.sol
+++ b/test/LiquidityPool.Deposits.t.sol
@@ -291,6 +291,9 @@ contract LiquidityPoolTest is TestSystem {
         // user_2 i-withdraw 20$
         // user_3 q-withdraw 13$

+        // Frontrun all withdrawal requests, since amount =0, c
+        pool.queueWithdraw(0, 0xDfA2d3a0d32F870D87f8A0d7AA6b9Cc
+
         vm.prank(user_1);
         pool.queueWithdraw(2e19, user_1);
         vm.prank(user_3);
```

Tools Used

Manual review, forge

Recommended Mitigation Steps

The processing of withdrawalQueue should have a mechanism to handle reverting `QueuedWithdraw` entries. Either by skipping them and/or moving them to another `failedWithdrawals` queue.

[mubaris (Polynomial) confirmed]

[Dravee (judge) commented]:

> The frontrunning part isn't an issue on Optimism, but the rest is valid.

[H-13] Exchange: `totalFunding` calculation should be done with a simple multiplication operation instead of a `wadMul` operation

Submitted by [carlitox477], also found by [KIntern_NA]

`wad` operations are meant to be done with int/uint which represent numbers with 18 decimals.

While the funding rate follows this representation, a simple time difference does not.

In `Exchange.getMarkPrice` as well as in `Exchange._updateFundingRate` a `wadMul` operation is done to multiply the funding rate per second by a simple time difference, leading to wrong calculation of `normalizationFactor` and mark price, affecting critical parts of the protocol.

Proof of Concept

In `Exchange.getMarkPrice` first we get the funding rate per second:

```
(int256 fundingRate,) = getFundingRate();
fundingRate = fundingRate / 1 days; // Funding rate per second
```

Immediately after, the total funding since last update is calculated:

```
int256 currentTimeStamp = int256(block.timestamp);
        int256 fundingLastUpdatedTimestamp = int256(fundingLastU

        // @audit funding rate X (time interval) / 1e18 = Number
```

```
int256 totalFunding = wadMul(fundingRate, (currentTimeSt
```

`int256 totalFunding = wadMul(fundingRate, (currentTimeStamp - fundingLastUpdatedTimestamp));`
is the same that
$TOTAL\_ FUNDING\_{t\_{1}; t\_{2}} = \frac{FUNDING\_ RATE\_{sec} \times (t\_{2} - t\_{1})}{10^{18}}$

However, the division by 10^{18} should not happen, given that time difference does not represent a number with 18 decimals.

This ends up in a miscalculation of `totalFunding` variable, and as a consequence, a miscalculation of mark price.

The same issue happens in `_updateFundingRate` function.

Impact

Here a complete list of functions affected by this bug:

1. `Exchange.\_updateFundingRate` (`Exchange.sol#L416`)

    1. `Exchange.openTrade` (`Exchange.sol#L95`)
    2. `Exchange.closeTrade` (`Exchange.sol#L108`)
    3. `Exchange.addCollateral` (`Exchange.sol#L121`)
    4. `Exchange.removeCollateral` (`Exchange.sol#L134`)
    5. `Exchange.liquidate` (`Exchange.sol#L147`)

2. `Exchange.getMarkPrice` (`Exchange.sol#L196`):

    1. `Exchange.\_addCollateral` (`Exchange.sol#L358`)
    2. `Exchange.\_removeCollateral` (`Exchange.sol#L384`)
    3. `KangarooVault.getTokenPrice` (`KangarooVault.sol#L353`)
    4. `ShortCollateral.liquidate` (`ShortCollateral.sol#L133`)

5. `ShortCollateral.getMinCollateral` (`ShortCollateral.sol#L163`)
6. `ShortCollateral.canLiquidate` (`ShortCollateral.sol#L193`)
7. `ShortCollateral.maxLiquidatableDebt` (`ShortCollateral.sol#L216`)
8. `LiquidityPool.orderFee` (`LiquidityPool.sol#L388`)
9. `LiquidityPool.getMarkPrice` (`LiquidityPool.sol#L405`)

    1. `LiquidityPool.getTokenPrice` (`LiquidityPool.sol#L352`)
    2. `LiquidityPool.openLong` (`LiquidityPool.sol#L437`)
    3. `LiquidityPool.closeLong`(`LiquidityPool.sol#L469`)
    4. `LiquidityPool.openShort` (`LiquidityPool.sol#L501`)
    5. `LiquidityPool.closeShort` (`LiquidityPool.sol#L533`)
    6. `LiquidityPool.liquidate` (`LiquidityPool.sol#L558`)
    7. `KangarooVault.removeCollateral` (`KangarooVault.sol#L437`)
    8. `KangarooVault.\_openPosition` (`KangarooVault.sol#L568`)

As it can be seen, this bug affects multiple critical part of the protocol, calculating the correct mark price as well as updating the funding rate is essential for the protocol correct behavior.

Recommended Mitigation steps

Simply replace current `wad` operation for a simple multiplication.

```
    function getMarkPrice() public view override returns (uint25
        // Get base asset price from oracles
        (uint256 baseAssetPrice, bool invalid) = pool.baseAssetI
        isInvalid = invalid;

        // Get funding rate per second
        // max 1% or 1e16
        (int256 fundingRate,) = getFundingRate();
        fundingRate = fundingRate / 1 days;

        int256 currentTimeStamp = int256(block.timestamp);
        int256 fundingLastUpdatedTimestamp = int256(fundingLastU
```

```diff
-        int256 totalFunding = wadMul(fundingRate, (currentTimeSt
+        int256 totalFunding = fundingRate, (currentTimeStamp - f
         int256 normalizationUpdate = 1e18 - totalFunding;
         uint256 newNormalizationFactor = normalizationFactor.mul

         uint256 squarePrice = baseAssetPrice.mulDivDown(baseAsse
         markPrice = squarePrice.mulWadDown(newNormalizationFacto
     }



     function _updateFundingRate() internal {
         (int256 fundingRate,) = getFundingRate();

         fundingRate = fundingRate / 1 days;

         int256 currentTimeStamp = int256(block.timestamp);
         int256 fundingLastUpdatedTimestamp = int256(fundingLastU

-        int256 totalFunding = wadMul(fundingRate, (currentTimeSt
+        int256 totalFunding = fundingRate, (currentTimeStamp - f

         int256 normalizationUpdate = 1e18 - totalFunding;

         normalizationFactor = normalizationFactor.mulWadDown(uin

         emit UpdateFundingRate(fundingLastUpdated, normalizatior
         fundingLastUpdated = block.timestamp;
     }
```

mubaris (Polynomial) confirmed

[H-14] Inexpedient liquidatable logic that could have half liquidable position turns fully liquidable instantly

Submitted by RaymondFam, *also found by* Josiah

In ShortCollateral.sol, the slash logic of `maxLiquidatableDebt()` is specifically too harsh to the barely unhealthy positions because `maxDebt` will be half of the position to be liquidated if `0.95e18 <= safetyRatio <= 1e18`.

Additionally, once a position turns liquidatable, the position is deemed fully liquidatable atomically in two repeated transactions.

Proof of Concept

Supposing we resort to the following setup as denoted in `ShortCollateral.t.sol` (`#L21-L23`):

collRatio = 1.5e18
liqRatio = 1.3e18
liqBonus = 1e17

Collateral ratio of a position, `x = (position.collateralAmount * collateralPrice) / (position.shortAmount * markPrice)`

File: `ShortCollateral.sol#L230-L239`

```
        uint256 safetyRatioNumerator = position.collateralAmount
        uint256 safetyRatioDenominator = position.shortAmount.mu
        safetyRatioDenominator = safetyRatioDenominator.mulWadDo
        uint256 safetyRatio = safetyRatioNumerator.divWadDown(sa

        if (safetyRatio > 1e18) return maxDebt;

        maxDebt = position.shortAmount / 2;

        if (safetyRatio < WIPEOUT_CUTOFF) maxDebt = position.sho
```

According to the code block above with `liqRatio` factored in:

In order to avoid being liquidated, a position will need to have a collateral ratio, `x > 1.3e18` so that `safetyRatio > (1.3 / 1.3)e18` which is `safetyRatio > 1e18`.

The position will be half liquidated if its associated collateral ratio falls in the range of `1.235e18 <= x <= 1.3e18`. To avoid full liquidation, the condition at the lower end will need to be `safetyRatio >= (1.235 / 1.3)e18` which is `safetyRatio >= 0.95e18`.

The position will be fully liquidated if `x < 1.235e18`.

Here is the unforgiving scenario:

1. Bob has a short position whose collateral ratio happens to be 1.3e18.
2. Bob's position gets half liquidated the first round ending up with a collateral ratio, x (Note: The numerator is multiplied by 0.45 because of the additional 10% `liqBonus` added to the one half collateral slashed:
   `(1.3 * 0.45 / 0.5)e18 = 1.17e18`
3. Immediately, Bob's position becomes fully liquidatable because `x < 1.235e18`.

Recommended Mitigation Steps

Consider restructuring the slashing logic such that the position turns healthy after being partially liquidated, instead of making it spiral down to the drain.

[Dravee (judge) commented](https://github.com/code-423n4/2023-03-polynomial-findings/issues/236#issuecomment-1483426553):

> Not a duplicate of https://github.com/code-423n4/2023-03-polynomial-findings/issues/146

[mubaris (Polynomial) confirmed]

[rivalq (Polynomial) confirmed and commented]:

> Depending upon the collateral and its collateral ratio etc, that spiral of liquidation may happen.

Medium Risk Findings (19)

[M-01] Exchange Rate can be manipulated if positions are big enough for a long enough time

Submitted by GalloDaSballo

If exchangeRate can be maniupulated, then this can be used to extract value or grief withdrawals from the `KangarooVault`.

From my experimentation, the values to manipulate the share price are very high, making the attack fairly unlikely.

That said, by manipulating `markPrice` we can get maniupulate `getTokenPrice`, which will cause a leak of value in withdrawals.

This requires a fairly laborious setup (enough time has passed, from fairly thorough testing we need at least 1 week).
And also requires a very high amount of capital (1 billion in the example, I think 100MLN works as well)

Proof of Concept

Can be run via `forge test --match-test testFundingRateDoesChange -vvvvv`

After creating a new test file `TestExchangeAttack.t.sol`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

import {console2} from "forge-std/console2.sol";

import {FixedPointMathLib} from "solmate/utils/FixedPointMathLik

import {TestSystem} from "./utils/TestSystem.sol";
import {Exchange} from "../src/Exchange.sol";
import {LiquidityPool} from "../src/LiquidityPool.sol";
import {PowerPerp} from "../src/PowerPerp.sol";
import {ShortToken} from "../src/ShortToken.sol";
import {ShortCollateral} from "../src/ShortCollateral.sol";
import {MockERC20Fail} from "../src/test-helpers/MockERC20Fail.s

contract TestExchangeAttack is TestSystem {
    using FixedPointMathLib for uint256;

    uint256 public constant initialPrice = 1200e18;

    Exchange private exchange;
```

```
    PowerPerp private powerPerp;
    ShortToken private shortToken;
    ShortCollateral private shortCollateral;
    MockERC20Fail private susd;

    LiquidityPool private pool;

    function setUp() public {
        deployTestSystem();
        initPool();
        initExchange();
        preparePool();
        setAssetPrice(initialPrice);

        exchange = getExchange();
        powerPerp = getPowerPerp();
        shortToken = getShortToken();
        shortCollateral = getShortCollateral();
        susd = getSUSD();
        pool = getPool();
    }


    function testDeployment() public {
        exchange.refresh();
        _testDeployment();
    }

    event Debug(string name, uint256 value);

    function testFundingRateDoesChange() public {
        (uint256 price,) = exchange.getIndexPrice();
        (uint256 markPrice,) = exchange.getMarkPrice();
        (int256 fundingRate,) = exchange.getFundingRate();

        assertEq(fundingRate, 0);

        deposit(1e40, user_3);

        // 10e26 = 1 Billion
        openLong(1e26,  1e34, user_2);
        // openShort(1e20,  user_1);

        vm.warp(block.timestamp + 20 weeks);

        (int256 newFundingRate,) = exchange.getFundingRate();
```

```
        emit Debug("fundingRate", uint256(fundingRate));
        emit Debug("newFundingRate", uint256(newFundingRate));
        assertTrue(fundingRate != newFundingRate, "newFundingRat

        (uint256 newMarkPrice,) = exchange.getMarkPrice();

        emit Debug("markPrice", markPrice);
        emit Debug("newMarkPrice", newMarkPrice);
        assertTrue(markPrice != newMarkPrice, "price has not cha
    }

    function deposit(uint256 amount, address user) internal {
        susd.mint(user, amount);

        vm.startPrank(user);
        susd.approve(address(getPool()), amount);
        pool.deposit(amount, user);
        vm.stopPrank();
    }
    function openLong(uint256 amount, uint256 maxCost, address u
        susd.mint(user, maxCost);

        Exchange.TradeParams memory tradeParams;

        tradeParams.isLong = true;
        tradeParams.amount = amount;
        tradeParams.maxCost = maxCost;

        vm.startPrank(user);
        susd.approve(address(getPool()), maxCost);
        exchange.openTrade(tradeParams);
        vm.stopPrank();
    }

    function closeLong(uint256 amount, uint256 minCost, address
        require(powerPerp.balanceOf(user) == amount);

        Exchange.TradeParams memory tradeParams;

        tradeParams.isLong = true;
        tradeParams.amount = amount;
        tradeParams.minCost = minCost;

        vm.prank(user);
        exchange.closeTrade(tradeParams);
```

```
    }

    function openShort(uint256 amount, address user)
        internal
        returns (uint256 positionId, Exchange.TradeParams memory
    {

        uint256 collateral = shortCollateral.getMinCollateral(am


        susd.mint(user, collateral);

        tradeParams.amount = amount;
        tradeParams.collateral = address(susd);
        tradeParams.collateralAmount = collateral;
        tradeParams.minCost = 0;


        vm.startPrank(user);
        susd.approve(address(exchange), collateral);
        (positionId,) = exchange.openTrade(tradeParams);
        vm.stopPrank();
    }

    function openShort(uint256 positionId, uint256 amount, uint2
        Exchange.TradeParams memory tradeParams;
        susd.mint(user, collateral);

        tradeParams.positionId = positionId;
        tradeParams.amount = amount;
        tradeParams.collateral = address(susd);
        tradeParams.collateralAmount = collateral;
        tradeParams.minCost = 0;

        vm.startPrank(user);
        susd.approve(address(exchange), collateral);
        (positionId,) = exchange.openTrade(tradeParams);
        vm.stopPrank();
    }

    function closeShort(uint256 positionId, uint256 amount, uint
        internal
        returns (Exchange.TradeParams memory tradeParams)
    {
        require(shortToken.ownerOf(positionId) == user);
        susd.mint(user, maxCost);
```

```
        (, uint256 shortAmount, uint256 collateralAmount, addres

        tradeParams.amount = amount > shortAmount ? shortAmount
        tradeParams.collateral = collateral;
        tradeParams.collateralAmount = collAmt > collateralAmour
        tradeParams.maxCost = maxCost;
        tradeParams.positionId = positionId;

        vm.startPrank(user);
        susd.approve(address(getPool()), maxCost);
        exchange.closeTrade(tradeParams);
        vm.stopPrank();
    }
}
```

As you can see we can move the markPrice, which will impact the valuation of the KangarooShares during withdrawals

```
    │   └─ ← 7199999999999999280, false
    ├─ emit Debug(name: markPrice, value: 7200000000000000000)
    ├─ emit Debug(name: newMarkPrice, value: 719999999999999992
    └─ ← ()
```

[mubaris (Polynomial) disagreed with severity](#)

[Dravee (judge) commented](#):

> Would like more input from the sponsor @mubaris on this one.

> Going back to the definitions:

> QA (Quality Assurance) Includes both Non-critical (code style, clarity, syntax, versioning, off-chain monitoring (events, etc) and Low risk (e.g. assets are not at risk: state handling, function incorrect as to spec, issues with comments). Excludes Gas optimizations, which are submitted and judged separately.
> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> 3 — High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).
> The attack path being an edge case on the assets that's valid but unlikely, I believe Medium Severity to still be right. Even if not "Sponsor Confirmed", this could be an acknowledged bug.

mubaris (Polynomial) confirmed and commented:

> Medium severity seems fair.

Dravee (judge) decreased severity to Medium

[M-02] Users can receive less collateral than expected from liquidations

Submitted by MiloTruck, also found by adriro, bin2chen, chaduke, and csanuragjain

Users might receive very little or no collateral when liquidating extremely unhealthy short positions.

Vulnerability Details

When users liquidate a short position, they expect to get a reasonable amount of collateral in return. The collateral amount sent to liquidators is handled by `liquidate()` in the `ShortCollateral` contract:

```
totalCollateralReturned = liqBonus + collateralClaim;
if (totalCollateralReturned > userCollateral.amount) totalCollat
userCollateral.amount -= totalCollateralReturned;

ERC20(userCollateral.collateral).safeTransfer(user, totalCollate
```

Where:

- `liqBonus` - Bonus amount of collateral for liquidation.
- `collateralClaim` - Collateral amount returned, proportional to the how much debt is being liquidated.

As seen from above, if the position does not have sufficient collateral to repay the short amount being liquidated, it simply repays the liquidator with the remaining collateral amount.

This could cause liquidators to receive less collateral than expected, especially if they fully liquidate positions with high short amount to collateral ratios. In extreme cases, if a user liquidates a position with a positive short amount and no collateral (known as bad debt), they would receive no collateral at all.

Proof of Concept

The following test demonstrates how a user can liquidate a short position without getting any collateral in return:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

import {TestSystem, Exchange, ShortToken, ShortCollateral, MockE

contract LiquidationOverpay is TestSystem {
    // Protocol contracts
    Exchange private exchange;
    ShortToken private shortToken;
    ShortCollateral private shortCollateral;

    // sUSD token contract
    MockERC20Fail private SUSD;

    // Intial base asset price
    uint256 private constant initialBaseAssetPrice = 1e18;

    function setUp() public {
        // Deploy contracts
        deployTestSystem();
        initPool();
        initExchange();
        preparePool();

        exchange = getExchange();
        shortToken = getShortToken();
        shortCollateral = getShortCollateral();
        SUSD = getSUSD();
```

```
        // Set initial price for base asset
        setAssetPrice(initialBaseAssetPrice);

        // Mint sUSD for user_1
        SUSD.mint(user_1, 1e20);

        // Mint powerPerp for user_2 and user_3
        vm.startPrank(address(exchange));
        getPowerPerp().mint(user_2, 1e20);
        getPowerPerp().mint(user_3, 1e20);
        vm.stopPrank();
    }

    function testLiquidationReturnsLessCollateralThanExpected()
        // Open short position with amount = 1e18, collateral an
        uint256 shortAmount = 1e18;
        uint256 positionId = openShort(1e18, 1e15, user_1);

        // Base asset price rises by 50%
        setAssetPrice(initialBaseAssetPrice * 150 / 100);

        // user_2 liquidates 85% USER's entire short position
        vm.prank(user_2);
        exchange.liquidate(positionId, shortAmount * 85 / 100);

        // positionId has no remaining collateral, but still has
        (, uint256 remainingAmount, uint256 remainingCollateralA
        assertEq(remainingAmount, shortAmount * 15 / 100);
        assertEq(remainingCollateralAmount, 0);

        // user_3 liquidates the same position
        vm.prank(user_3);
        exchange.liquidate(positionId, shortAmount);

        // user_3 did not get any collateral
        assertEq(SUSD.balanceOf(user_3), 0);
    }

    function openShort(
        uint256 amount,
        uint256 collateralAmount,
        address user
    ) internal returns (uint256 positionId) {
        Exchange.TradeParams memory tradeParams;
        tradeParams.amount = amount;
        tradeParams.collateral = address(SUSD);
```

```
        tradeParams.collateralAmount = collateralAmount;

        vm.startPrank(user);
        SUSD.approve(address(exchange), collateralAmount);
        (positionId, ) = exchange.openTrade(tradeParams);
        vm.stopPrank();
    }
}
```

Recommended Mitigation

In `liquidate()` of the `Exchange` contract, consider adding a `minCollateralAmount` parameter, which represents the minimum amount of collateral a liquidator is willing to receive. . If the returned collateral amount is less than `minCollateralAmount`, the transaction should revert.

[Dravee (judge) commented](#):

> Warden wrote:
>
> ```
>         // Base asset price rises by 50%
>         setAssetPrice(initialBaseAssetPrice * 150 / 100);
> ```
>
> Base asset being sUSD, not sure how likely it is to rise by 50%. Crazy world though and fuzzed tests DO take into account such an increase:

```
File: Exchange.Simple.t.sol
188:     function testSimpleLongClosePriceDiffTrade(uint256 pric
189:         vm.assume(priceDiff > 1e17);
190:         vm.assume(priceDiff < 50e18);
...
196:         setAssetPrice(initialPrice + priceDiff);
```

> Seems valid.

[mubaris (Polynomial) confirmed](#)

[M-03] `KangarooVault.initiateDeposit`, `KangarooVault.processDepositQueue`, `KangarooVault.initiateWithdrawal`, and `KangarooVault.processWithdrawalQueue` functions do not use `whenNotPaused` modifier

Submitted by [rbserver](), *also found by* [DadeKuma](), [CRYP70](), [sakshamguruji](), *and* [Diana]()

As shown by the code below, although `PauseModifier` is imported, the `KangarooVault` contract does not use the `whenNotPaused` modifier in any of its functions. More specifically, the `KangarooVault.initiateDeposit`, `KangarooVault.processDepositQueue`, `KangarooVault.initiateWithdrawal`, and `KangarooVault.processWithdrawalQueue` functions do not use the `whenNotPaused` modifier.

```
import {PauseModifier} from "./utils/PauseModifier.sol";

contract KangarooVault is Auth, ReentrancyGuard, PauseModifier {


    function initiateDeposit(address user, uint256 amount) exter


    function processDepositQueue(uint256 idCount) external nonRe


    function initiateWithdrawal(address user, uint256 tokens) ex


    function processWithdrawalQueue(uint256 idCount) external nc
```

This is unlike the `LiquidityPool` contract; comparing to the `KangarooVault.initiateDeposit`, `KangarooVault.processDepositQueue`, `KangarooVault.initiateWithdrawal`, and

`KangarooVault.processWithdrawalQueue` functions, the `LiquidityPool.deposit`, `LiquidityPool.queueDeposit`, `LiquidityPool.processDeposits`, `LiquidityPool.withdraw`, `LiquidityPool.queueWithdraw`, and `LiquidityPool.processWithdraws` functions have the similar functionalities but they all use the `whenNotPaused` modifier. As a result, when an emergency, such as a hack, occurs, the protocol can pause the `LiquidityPool.withdraw`, `LiquidityPool.queueWithdraw`, and `LiquidityPool.processWithdraws` functions to prevent or reduce damages, such as preventing users and the protocol from losing funds, but cannot do that for the `KangarooVault.initiateDeposit`, `KangarooVault.processDepositQueue`, `KangarooVault.initiateWithdrawal`, and `KangarooVault.processWithdrawalQueue` functions.

```
    function deposit(uint256 amount, address user) external over


    function queueDeposit(uint256 amount, address user)
        external
        override
        nonReentrant
        whenNotPaused("POOL_QUEUE_DEPOSIT")
    {


    function processDeposits(uint256 count) external override no


    function withdraw(uint256 tokens, address user) external ove


    function queueWithdraw(uint256 tokens, address user)
        external
        override
        nonReentrant
        whenNotPaused("POOL_QUEUE_WITHDRAW")
    {
```

```
function processWithdraws(uint256 count) external override r
```

Proof of Concept

The following steps can occur for the described scenario.

1. An emergency, such as a hack, occurs in which further withdrawals can cause users and the protocol to lose funds.
2. The protocol team is able to pause the `LiquidityPool.withdraw`, `LiquidityPool.queueWithdraw`, and `LiquidityPool.processWithdraws` functions.
3. However, the protocol team is unable to pause the `KangarooVault.initiateWithdrawal` and `KangarooVault.processWithdrawalQueue` functions.
4. As a result, funds can be lost from the `KangarooVault`.

Tools Used

VSCode

Recommended Mitigation Steps

The `KangarooVault.initiateDeposit`, `KangarooVault.processDepositQueue`, `KangarooVault.initiateWithdrawal`, and `KangarooVault.processWithdrawalQueue` functions can be updated to use the `whenNotPaused` modifier.

[rivalq (Polynomial) disagreed with severity]()

[Dravee (judge) commented]():

> rivalq (Polynomial) disagreed with severity

> Not such an easy one to grade. Historically, missing `whenNotPaused` modifiers are considered as a medium risk issue:

- [https://github.com/code-423n4/2022-06-connext-findings/issues/175]()

- https://github.com/code-423n4/2022-09-y2k-finance-findings/issues/38

> The only time they are considered as Low severity findings are when they're actually put on too many functions and should be removed:

- https://code4rena.com/reports/2022-02-jpyc#l-04-fiattokenv1—v2-remove-whennotpaused-modifier-from-cancelauthorization-and-decreaseallowance-functions
- https://code4rena.com/reports/2023-01-drips#l-04-collecting-funds-should-be-usable-while-the-dripshub-contract-is-paused

> Therefore I believe that Medium Severity is the right risk level, at least for this type of issue on code4rena and for the time being

mubaris (Polynomial) confirmed and commented:

> Medium severity seems fair.

[M-04] `LiquidityPool._getDelta` and `LiquidityPool._calculateMargin` functions should execute `require(!isInvalid && spotPrice > 0)` instead of `require(!isInvalid || spotPrice > 0)`

Submitted by rbserver, also found by juancito, peakbolt, and chaduke

The following `KangarooVault.transferPerpMargin` and `KangarooVault._openPosition` functions execute `require(!isInvalid && baseAssetPrice != 0)`, where `isInvalid` and `baseAssetPrice` are returned from calling `LIQUIDITY_POOL.baseAssetPrice()`. Because `LIQUIDITY_POOL.baseAssetPrice()` returns the return values of `perpMarket.assetPrice()`, the `KangarooVault.transferPerpMargin` and `KangarooVault._openPosition` functions would only consider `perpMarket.assetPrice()` as reliable when both `!isInvalid` and `baseAssetPrice != 0` are true.

```
    function transferPerpMargin(int256 marginDelta) external rec
        if (marginDelta < 0) {
```

```
            ...
            (uint256 baseAssetPrice, bool isInvalid) = LIQUIDITY
            require(!isInvalid && baseAssetPrice != 0);
            ...
        } else {
            ...
        }
        ...
    }
```

```
    function _openPosition(uint256 amt, uint256 minCost) interna
        ...
        (uint256 baseAssetPrice, bool isInvalid) = LIQUIDITY_POO
        require(!isInvalid && baseAssetPrice != 0);
        ...
    }
```

However, the following `LiquidityPool._getDelta` and `LiquidityPool._calculateMargin` functions execute `require(!isInvalid || spotPrice > 0)` and would consider `perpMarket.assetPrice()` as reliable when either `!isInvalid` or `spotPrice > 0` is true. When `perpMarket.assetPrice()` returns a positive `spotPrice` and a true `isInvalid`, such `spotPrice` should be considered as invalid and untrusted; trades, such as these for opening and closing long and short positions using the `LiquidityPool`, that depend on the `LiquidityPool._getDelta` and `LiquidityPool._calculateMargin` functions' return values, which then rely on such invalid `spotPrice`, should not be allowed. However, because `!isInvalid || spotPrice > 0` is true in this case, calling the `LiquidityPool._getDelta` and `LiquidityPool._calculateMargin` functions will not revert, and such trades that should not be allowed can still be made.

```
    function _getDelta() internal view returns (uint256 delta) {
        (uint256 spotPrice, bool isInvalid) = baseAssetPrice();
        uint256 pricingConstant = exchange.PRICING_CONSTANT();

        require(!isInvalid || spotPrice > 0);

        delta = spotPrice.mulDivDown(2e18, pricingConstant);
        delta = delta.mulWadDown(exchange.normalizationFactor())
```

```
    }


    function _calculateMargin(int256 size) internal view returns
        (uint256 spotPrice, bool isInvalid) = baseAssetPrice();

        require(!isInvalid || spotPrice > 0);

        uint256 absSize = size.abs();
        margin = absSize.mulDivDown(spotPrice, futuresLeverage);
    }


    function baseAssetPrice() public view override returns (uint
        (spotPrice, isInvalid) = perpMarket.assetPrice();
    }
```

Proof of Concept

The following steps can occur for the described scenario.

1. Calling the `perpMarket.assetPrice` function returns a positive `spotPrice` and a true `isInvalid` at this moment.
2. Calling the `LiquidityPool._getDelta` and `LiquidityPool._calculateMargin` functions would not revert because `require(!isInvalid || spotPrice > 0)` would be passed.
3. Trades, such as for closing a long position through calling the `LiquidityPool.closeLong` function, can go through even though the used `spotPrice` is invalid and untrusted.
4. In this case, such trades should not be allowed but are still made.

Tools Used

VSCode

Recommended Mitigation Steps

The `LiquidityPool._getDelta` and `LiquidityPool._calculateMargin` functions can be updated to execute `require(!isInvalid && spotPrice > 0)` instead of `require(!isInvalid || spotPrice > 0)`.

mubaris (Polynomial) confirmed

[M-05] Some functions that call `Exchange.getMarkPrice` function do not check if `Exchange.getMarkPrice` function's returned `markPrice` is 0

Submitted by rbserver

The following `Exchange.getMarkPrice` function uses `pool.baseAssetPrice()`'s returned `baseAssetPrice`, which is `spotPrice` returned by `perpMarket.assetPrice()`, to calculate and return the `markPrice`. When such `spotPrice` is 0, this function would return a 0 `markPrice`.

```
function getMarkPrice() public view override returns (uint25
    (uint256 baseAssetPrice, bool invalid) = pool.baseAssetF
    isInvalid = invalid;

    (int256 fundingRate,) = getFundingRate();
    fundingRate = fundingRate / 1 days;

    int256 currentTimeStamp = int256(block.timestamp);
    int256 fundingLastUpdatedTimestamp = int256(fundingLastU

    int256 totalFunding = wadMul(fundingRate, (currentTimeSt
    int256 normalizationUpdate = 1e18 - totalFunding;
    uint256 newNormalizationFactor = normalizationFactor.mul

    uint256 squarePrice = baseAssetPrice.mulDivDown(baseAsse
    markPrice = squarePrice.mulWadDown(newNormalizationFacto
}
```

```
function baseAssetPrice() public view override returns (uint
    (spotPrice, isInvalid) = perpMarket.assetPrice();
}
```

As shown by the code below, calling the `KangarooVault.transferPerpMargin` and `KangarooVault._openPosition` functions would revert if `baseAssetPrice` returned by `LIQUIDITY_POOL.baseAssetPrice()` is 0 no matter what the returned `isInvalid` is. This means that the price returned by `perpMarket.assetPrice()` should not be trusted and used whenever such price is 0.

```
function transferPerpMargin(int256 marginDelta) external rec
    if (marginDelta < 0) {
        ...
        (uint256 baseAssetPrice, bool isInvalid) = LIQUIDITY
        require(!isInvalid && baseAssetPrice != 0);
        ...
    } else {
        ...
    }
    ...
}
```

```
function _openPosition(uint256 amt, uint256 minCost) interna
    ...
    (uint256 baseAssetPrice, bool isInvalid) = LIQUIDITY_POO
    require(!isInvalid && baseAssetPrice != 0);
    ...
}
```

However, some functions that call the `Exchange.getMarkPrice` function do not additionally check if the `Exchange.getMarkPrice` function's returned `markPrice` is 0, which can lead to unexpected consequences. For example, the following `KangarooVault.removeCollateral` function executes `(uint256 markPrice,) = LIQUIDITY_POOL.getMarkPrice()`. When `markPrice` is 0, which is caused by a 0 `spotPrice` returned by `perpMarket.assetPrice()`, such price should be considered as invalid and should not be used; yet, in this case, such 0 `markPrice` can cause `minColl` to also be 0, which then makes `require(positionData.totalCollateral >= minColl + collateralToRemove)` much more likely to be passed. In this situation, calling the `KangarooVault.removeCollateral` function can remove the specified `collateralToRemove` collateral from the Power Perp position but this actually should not be allowed because such 0 `spotPrice` and 0 `markPrice` should be considered as invalid and should not be used.

```
function removeCollateral(uint256 collateralToRemove) extern
    (uint256 markPrice,) = LIQUIDITY_POOL.getMarkPrice();
    uint256 minColl = positionData.shortAmount.mulWadDown(ma
    minColl = minColl.mulWadDown(collRatio);

    require(positionData.totalCollateral >= minColl + collat

    usedFunds -= collateralToRemove;
    positionData.totalCollateral -= collateralToRemove;

    emit RemoveCollateral(positionData.positionId, collatera
}
```

```
function getMarkPrice() public view override returns (uint25
    return exchange.getMarkPrice();
}
```

Proof of Concept

The following steps can occur for the described scenario.

1. The `KangarooVault.removeCollateral` function is called with `collateralToRemove` being 100e18.
2. `markPrice` returned by `LIQUIDITY_POOL.getMarkPrice()` is 0 because a 0 `spotPrice` is returned by `perpMarket.assetPrice()` .
3. Due to the 0 `markPrice` , `minColl` is 0, and `positionData.totalCollateral >= minColl + collateralToRemove` can be true even if `positionData.totalCollateral` is also 100e18 at this moment.
4. Calling the `KangarooVault.removeCollateral` function does not revert, and 100e18 collateral is remove from the Power Perp position.
5. However, a 0 `spotPrice` returned by `perpMarket.assetPrice()` and a 0 `markPrice` returned by `LIQUIDITY_POOL.getMarkPrice()` should be considered as invalid and should not be used. In this case, removing 100e18

collateral from the Power Perp position should not be allowed or succeed but it does.

Tools Used

VSCode

Recommended Mitigation Steps

Functions, such as the `KangarooVault.removeCollateral` function, that call the `Exchange.getMarkPrice` function can be updated to additionally check if the `Exchange.getMarkPrice` function's returned `markPrice` is 0. If it is 0, calling these functions should revert.

[mubaris (Polynomial) confirmed]

[M-06] `KangarooVault._resetTrade`, `LiquidityPool.rebalanceMargin`, and `LiquidityPool._getTotalMargin` functions should check `isInvalid`, which is returned by external perp market contract's `remainingMargin` function, and revert if such `isInvalid` is true

Submitted by [rbserver], also found by [KIntern_NA]

The following `KangarooVault._resetTrade`, `LiquidityPool.rebalanceMargin`, and `LiquidityPool._getTotalMargin` functions call the external perp market contract's `remainingMargin` function to get the remaining margin for the `KangarooVault` or `LiquidityPool`. Yet, none of these functions use `isInvalid` that can be returned by such `remainingMargin` function. When the returned remaining margin is invalid, such value should not be trusted and used. Using such invalid remaining margin can have negative effects.

```
    function _resetTrade() internal {
        positionData.positionId = 0;
        (uint256 totalMargin,) = PERP_MARKET.remainingMargin(adc
```

```
        PERP_MARKET.transferMargin(-int256(totalMargin));
        usedFunds -= totalMargin;
        ...
    }
```

```
    function rebalanceMargin(int256 marginDelta) external requir
        int256 currentPosition = _getTotalPerpPosition();
        uint256 marginRequired = _calculateMargin(currentPositic
        (uint256 currentMargin,) = perpMarket.remainingMargin(ac

        int256 additionalMargin = marginDelta;

        if (currentMargin >= marginRequired) {
            marginDelta -= int256(currentMargin - marginRequirec
        } else {
            marginDelta += int256(marginRequired - currentMargir
        }
        ...
    }
```

```
    function _getTotalMargin() internal view returns (uint256) {
        (uint256 margin,) = perpMarket.remainingMargin(address(t

        return margin;
    }
```

For example, we can compare the `KangarooVault.getTokenPrice` and `LiquidityPool.getTokenPrice` functions. The following `KangarooVault.getTokenPrice` function checks the `isInvalid` returned by `PERP_MARKET.remainingMargin(address(this))` so calling it will revert if `PERP_MARKET.remainingMargin(address(this))`'s returned `totalMargin` is invalid. In contrast, the `LiquidityPool.getTokenPrice` function does not do this.

```
    function getTokenPrice() public view returns (uint256) {
        ...
        (totalMargin, isInvalid) = PERP_MARKET.remainingMargin(a
        require(!isInvalid);
        ...
```

```
}
```

In the following `LiquidityPool.getTokenPrice` function, when `_getTotalMargin()` 's returned `totalMargin` , which is also `margin` returned by `perpMarket.remainingMargin(address(this))` in the `LiquidityPool._getTotalMargin` function, is invalid, such `totalMargin` is still used to increase `totalValue` , which then affects the liquidity token's price. The `LiquidityPool.getTokenPrice` function is called in functions like `LiquidityPool.deposit` and `LiquidityPool.withdraw` . Thus, calling such functions would not revert when such invalid `totalMargin` is used. As a result, the depositing and withdrawal actions that should not be allowed because of such invalid `totalMargin` can still be allowed unexpectedly.

```
function getTokenPrice() public view override returns (uint2
    if (totalFunds == 0) {
        return 1e18;
    }

    uint256 totalSupply = liquidityToken.totalSupply() + tot
    int256 skew = _getSkew();

    if (skew == 0) {
        return totalFunds.divWadDown(totalSupply);
    }

    (uint256 markPrice, bool isInvalid) = getMarkPrice();
    require(!isInvalid);

    uint256 totalValue = totalFunds;

    uint256 amountOwed = markPrice.mulWadDown(powerPerp.tota
    uint256 amountToCollect = markPrice.mulWadDown(shortToke
    uint256 totalMargin = _getTotalMargin();

    totalValue += totalMargin + amountToCollect;
    totalValue -= uint256((int256(amountOwed) + usedFunds));

    return totalValue.divWadDown(totalSupply);
}
```

Proof of Concept

The following steps can occur for the described scenario.

1. The `LiquidityPool.deposit` function is called to deposit some sUSD tokens.
2. When the `LiquidityPool.getTokenPrice` function is called, `_getTotalMargin()` 's returned `totalMargin` , which is `margin` returned by `perpMarket.remainingMargin(address(this))` , is invalid. However, such invalid `totalMargin` does not cause calling the `LiquidityPool.getTokenPrice` function to revert.
3. The deposit action succeeds while the used liquidity token's price that depends on the invalid `totalMargin` is inaccurate. Such deposit action should not be allowed but it is.

Tools Used

VSCode

Recommended Mitigation Steps

The `KangarooVault._resetTrade` , `LiquidityPool.rebalanceMargin` , and `LiquidityPool._getTotalMargin` functions can be updated to check `isInvalid` that is returned by the external perp market contract's `remainingMargin` function. If such `isInvalid` is true, calling these functions should revert.

[mubaris (Polynomial) confirmed]

[M-07] `usedFunds` can be greater than `totalFunds` in contract `KangarooVault` , which leads to `KangarooVault` being unable to close its trades and users being unable to withdraw

Submitted by [KIntern_NA]

`KangarooVault` contract can't close the trades and users can't withdraw from it, then users and `KangarooVaults` will lose a lot of funds.

Proof of concept

1. In contract `KangarooVault`, `usedFunds` is the uint256 variable which tracks the funds utilitized from the vault. And `totalFunds` is the uin256 variable which tracks the funds claimed by vault from profits and users' depositing.
2. Contract `KangarooVault` has no check if `totalFunds >= usedFunds` when `usedFunds` is increased (transfer from vault) or `totalFunds` is decreased (transfer to vault).
3. If someone transfers funds directly into the vault, `usedFunds` can be greater than `totalFunds` because the vault can transfer out more than `totalFunds`.
4. When `usedFunds > totalFunds`, KangarooVault can not close its trades because it will revert on underflow in function `_resetTrade`:

```
function _resetTrade() internal {
        ...
        totalFunds -= usedFunds;
        ...
    }
```

5. When `usedFunds > totalFunds`, user can't not withdraw by function `processWithdrawalQueue` because it will revert on underflow.

```
function processWithdrawalQueue(uint256 idCount) external nonRee
    for (uint256 i = 0; i < idCount; i++) {
        ...

        uint256 availableFunds = totalFunds - usedFunds;

        ...
```

Scenario:

- A deposit 1000 SUSD, then `totalFunds` = 1000 SUSD, `usedFunds` = 0
- B transfer directly 1000 SUSD to the KangarooVault, that doesn't change `totalFunds` and `usedFunds`

- KangarooVault opens a short position and uses 2000 SUSD to increase the margin, then `usedFunds` = 2000
- After that, `usedFunds > totalFunds` (2000 > 1000) then KangarooVault can't close its position

Recommended Mitigation Steps

Should add the checks if `totalFunds >= usedFunds` when increasing `usedfunds` or decreasing `totalFunds` in contract `KangarooVault.sol`.

mubaris (Polynomial) acknowledged, but disagreed with severity

Dravee (judge) decreased severity to Medium

[M-08] LiquidityPool can be DoS when a complete withdrawal is performed

Submitted by peakbolt, also found by auditor0517

`LiquidityPool` can be DoS when all funds are withdrawn from the pool, causing `getTokenPrice()` to revert due to zero `totalSupply()`.

Impact

The `LiquidityPool` will not be able to proceed and requires new deployment of the contracts.

Detailed Explanation

The `LiquidityPool.getTokenPrice()` will encounter a division by zero error when the `totalSupply` is zero but the `totalFunds` is non-zero. This could occur when there are partial withdrawals, causing totalFunds to be non-zero after a complete withdrawal, due to rounding from the withdrawals.

```
    function getTokenPrice() public view override returns (uint256)
        if (totalFunds == 0) {
            return 1e18;
        }
```

```
        uint256 totalSupply = liquidityToken.totalSupply() + totalQu
        int256 skew = _getSkew();

        if (skew == 0) {
            return totalFunds.divWadDown(totalSupply);
        }

        (uint256 markPrice, bool isInvalid) = getMarkPrice();
        require(!isInvalid);

        uint256 totalValue = totalFunds;

        uint256 amountOwed = markPrice.mulWadDown(powerPerp.totalSup
        uint256 amountToCollect = markPrice.mulWadDown(shortToken.to
        uint256 totalMargin = _getTotalMargin();

        totalValue += totalMargin + amountToCollect;
        totalValue -= uint256((int256(amountOwed) + usedFunds));

        return totalValue.divWadDown(totalSupply);
    }
```

Proof of Concept

Add the following test case to `test/LiquidityPool.Trades.t.sol`

```
    function testLiquidityPoolTokenPriceError() public {
        uint256 longAmount = 100e18;
        int256 skew = pool.getSkew();
        assertEq(skew, 0);

        // make some trades for pool to earn fees
        susd.mint(user_1, 100e18);
        openLong(longAmount, longAmount * 1000, user_1);
        setAssetPrice(initialPrice * 99 / 100);
        closeLong(longAmount, 0, user_1);

        // Queue withdrawal. This was deposited in TestSystem.prepar
        pool.queueWithdraw(1_000_000e18, address(this));
        skip(14500);

        // This will perform a partial withdrawals due to margin in
        // we use this just to trigger some rounding so that
```

```
        // totalFunds wont be zero during we fully withdraw from the
        pool.processWithdraws(1);

        // Transfer back margins from perpMarket so that
        // we can continue to withdraw the balance.
        pool.rebalanceMargin(-705672000000000000000);

        // This will do a complete withdrawals, reducing totalSupply
        // However, getTokenPrice() will encounter division by zero
        // That is because there will be small amount of totalFund l
        pool.processWithdraws(1);

        // approve funds for deposit again
        susd.approve(address(pool), 1_000_000e18);

        // This deposit will revert as LiquidityPool.getTokenPrice()
        // will encounter division by zero error.
        vm.expectRevert();
        pool.deposit(1_000_000e18, address(this));

    }
```

Recommended Mitigation Steps

Add in validation to check and handle when totalSupply is zero.

[mubaris (Polynomial) confirmed](#)

[M-09] Short positions with minimum collateral can be liquidated even though `canLiquidate()` returns `false`

Submitted by [MiloTruck](#), also found by [joestakey](#) and [Nyx](#)

Frontends or contracts that rely on `canLiquidate()` to determine if a position is liquidatable could be incorrect. Users could think their positions are safe from liquidation even though they are liquidatable, leading to them losing their collateral.

Vulnerability Details

In the `ShortCollateral` contract, `canLiquidate()` determines if a short position can be liquidated using the following formula:

```
uint256 minCollateral = markPrice.mulDivUp(position.shortAmount,
minCollateral = minCollateral.mulWadDown(collateral.liqRatio);

return position.collateralAmount < minCollateral;
```

Where:

- `position.collateralAmount` - Amount of collateral in the short position.
- `minCollateral` - Minimum amount of collateral required to avoid liquidation.

From the above, a short position can be liquidated if its collateral amount is **less than** `minCollateral`. This means a short position with the minimum collateral amount (ie. `position.collateralAmount == minCollateral`) cannot be liquidated.

However, this is not the case in `maxLiquidatableDebt()`, which is used to determine a position's maximum liquidatable debt:

```
uint256 safetyRatioNumerator = position.collateralAmount.mulWadI
uint256 safetyRatioDenominator = position.shortAmount.mulWadDowr
safetyRatioDenominator = safetyRatioDenominator.mulWadDown(colla
uint256 safetyRatio = safetyRatioNumerator.divWadDown(safetyRati

if (safetyRatio > 1e18) return maxDebt;

maxDebt = position.shortAmount / 2;
```

Where:

- `safetyRatio` - Equivalent to `position.collateralAmount / minCollateral`. Can be seen as a position's collateral amount against the minimum collateral required.
- `maxDebt` - The amount of debt liquidatable. Defined as 0 at the start of the function.

As seen from the `safetyRatio > 1e18` check, a position is safe from liquidation (ie. `maxDebt = 0`) if its `safetyRatio` is **greater than** 1.

Therefore, as a position with the minimum collateral amount has a `safetyRatio` of 1, half its debt becomes liquidatable. This contradicts `canLiquidate()`, which returns `false` for such positions.

Proof of Concept

The following test demonstrates how a position with minimum collateral is liquidatable even though `canLiquidate()` returns `false`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

import {TestSystem, Exchange, ShortToken, ShortCollateral, Power

contract CanLiquidateIsInaccurate is TestSystem {
    // Protocol contracts
    Exchange private exchange;
    ShortToken private shortToken;
    ShortCollateral private shortCollateral;

    // sUSD token contract
    MockERC20Fail private SUSD;

    function setUp() public {
        // Set liquidation ratio of sUSD to 125%
        susdLiqRatio = 1.24e18;

        // Deploy contracts
        deployTestSystem();
        initPool();
        initExchange();
        preparePool();

        exchange = getExchange();
        shortToken = getShortToken();
        shortCollateral = getShortCollateral();
        SUSD = getSUSD();

        // Mint sUSD for user_1
        SUSD.mint(user_1, 1e20);

        // Mint powerPerp for user_2
        vm.prank(address(exchange));
```

```
        getPowerPerp().mint(user_2, 1e20);
    }

    function testCanLiquidateMightBeWrong() public {
        // Initial price of base asset is 1e18
        uint256 initialPrice = 1e18;
        setAssetPrice(initialPrice);

        // Open short position with 1e15 sUSD as collateral
        Exchange.TradeParams memory tradeParams;
        tradeParams.amount = 1e18;
        tradeParams.collateral = address(SUSD);
        tradeParams.collateralAmount = 1e15;
        tradeParams.minCost = 0;

        vm.startPrank(user_1);
        SUSD.approve(address(exchange), tradeParams.collateralAm
        (uint256 positionId,) = exchange.openTrade(tradeParams);
        vm.stopPrank();

        // Initial price of base asset increases, such that min(
        setAssetPrice(1270001270001905664);

        // canLiquidate() returns false
        assertFalse(shortCollateral.canLiquidate(positionId));

        // However, maxLiquidatableDebt() returns half of origir
        assertEq(shortCollateral.maxLiquidatableDebt(positionId)

        // Other users can liquidate the short position
        vm.prank(user_2);
        exchange.liquidate(positionId, tradeParams.amount);

        // Position's shortAmount and collateral is reduced
        (, uint256 remainingAmount, uint256 remainingCollateralA
        assertEq(remainingAmount, tradeParams.amount / 2);
        assertLt(remainingCollateralAmount, tradeParams.collater
    }
}
```

Recommended Mitigation

Consider making short positions safe from liquidation if their `safetyRatio` equals to 1:

`ShortCollateral.sol#L235`:

```diff
-        if (safetyRatio > 1e18) return maxDebt;
+        if (safetyRatio >= 1e18) return maxDebt;
```

[mubaris (Polynomial) confirmed]

[M-10] Malicious users can exploit deposit and withdrawal queueing in `KangarooVault` and `LiquidityPool` contracts to force exorbitant transaction fees

Submitted by [bytes032], *also found by* [peanuts], [juancito], [sorrynotsorry], [0xbepresent], [PaludoX0], *and* [0xbepresent]

Direct loss of money for users who have deposited funds and wish to withdraw them, as they would be required to pay extremely high gas fees due to the queuing mechanism exploited by the attacker.

Proof of Concept

The affected contracts, `KangarooVault` and `LiquidityPool`, provide the functionality for instant deposits and withdrawals of tokens. However, in certain scenarios, deposits and withdrawals can be queued.

In both cases, the deposit transfers the `sUSD` from your address immediately. In the instant deposit scenario,mints you Vault or Liquidity token - depending on the contract. On the other hand, queuing it adds you to the deposit queue, by creating a "QueuedDeposit" object and sets its id to the current value of `nextQueuedDepositId` (mentioning that, because its going to be important later)

```
            QueuedDeposit storage newDeposit = depositQueue[nextQueu
            newDeposit.id = nextQueuedDepositId++;
            newDeposit.user = user;
            newDeposit.depositedAmount = amount;
            newDeposit.requestedTime = block.timestamp;
            totalQueuedDeposits += amount;
```

In `KangarooVault`, the deposits are queued ONLY If there are currently active positions.

```
function initiateDeposit(address user, uint256 amount) exter
    require(user != address(0x0));
    require(amount >= minDepositAmount);

    // Instant processing
    if (positionData.positionId == 0) {
        uint256 tokenPrice = getTokenPrice();
        uint256 tokensToMint = amount.divWadDown(tokenPrice)
        VAULT_TOKEN.mint(user, tokensToMint);
        totalFunds += amount;
        emit ProcessDeposit(0, user, amount, tokensToMint, b
    } else {
        // Queueing the deposit request
        QueuedDeposit storage newDeposit = depositQueue[next

        newDeposit.id = nextQueuedDepositId++;
        newDeposit.user = user;
        newDeposit.depositedAmount = amount;
        newDeposit.requestedTime = block.timestamp;

        totalQueuedDeposits += amount;
        emit InitiateDeposit(newDeposit.id, msg.sender, user
    }

    // SUSD checks for zero address
    SUSD.safeTransferFrom(msg.sender, address(this), amount)
}
```

On the other hand, in `LiquidityPool` (`LiquidityPool.sol#L201-L216`) instant deposits have a fee whereas the regular deposits don't have a fee. After discussing with the protocol team, queue deposits its primarily for the regular traders, because the price won't fluctate that much and instant is mostly for other protocols to build on top.

```
function queueDeposit(uint256 amount, address user)
    external
    override
    nonReentrant
```

```
    whenNotPaused("POOL_QUEUE_DEPOSIT")
{
    QueuedDeposit storage newDeposit = depositQueue[nextQueu
    newDeposit.id = nextQueuedDepositId++;
    newDeposit.user = user;
    newDeposit.depositedAmount = amount;
    newDeposit.requestedTime = block.timestamp;
    totalQueuedDeposits += amount;
    SUSD.safeTransferFrom(msg.sender, address(this), amount)

    emit InitiateDeposit(newDeposit.id, msg.sender, user, an
}
```

Finally, the deposits are processed in a nearly identical way (`LiquidityPool` , `KangarooVault`). Below, I've only extracted the vulnerable code which contains a for loop that iterates through a specified number of deposits, denoted by the variable `count` . The main purpose of this loop is to process each deposit in the queue and update the overall state of the contract.

At the beginning of each iteration, the function accesses the deposit at the current `queuedDepositHead` position in the `depositQueue` . The deposit's `requestedTime` is then evaluated to ensure that it is not equal to 0 and that the current block timestamp exceeds the deposit's `requestedTime` plus the `minDepositDelay` . If either of these conditions is true, the function terminates early, halting further deposit processing.

Upon passing the aforementioned check, it mints the tokens for the user and updates the contract accounting balance. Finally, the `queuedDepositHead` is incremented, advancing to the next deposit in the queue.

We can conclude that:

- `queuedDepositHead` = the next in line deposit to be executed
- `nextQueuedDepositId` = currently, the last deposit id in the queue

As a result, this means all the deposits are executed in sequential order and if there are currently 10 deposits and yours is the 11th, if you want to process your own, you have to process the 10 before that or wait for somebody else to process them.

```
        for (uint256 i = 0; i < count; i++) {
            QueuedDeposit storage current = depositQueue[queuedI

            if (current.requestedTime == 0 || block.timestamp <
                return;
            }

            uint256 tokensToMint = current.depositedAmount.divWa

            current.mintedTokens = tokensToMint;
            totalQueuedDeposits -= current.depositedAmount;
            totalFunds += current.depositedAmount;
            liquidityToken.mint(current.user, tokensToMint);

            emit ProcessDeposit(current.id, current.user, currer

            current.depositedAmount = 0;
            queuedDepositHead++;
        }
```

The queuing mechanism can be exploited by a malicious actor, who can queue a large number of deposits or withdrawals for a very low cost. This essentially locks all the deposited funds in the contract and forces the users to pay extremely high gas fees to process their transactions. This vulnerability can be exploited in a similar way for both contracts.

This can be mitigated to extent by the `minDepositAmount` variable, but that will just make the attack vector a bit more expensive for the attacker and the vulnerability would still be there.

To apply that to real world, assume the following scenario:

1. Alice queues 10000 deposits for 1 wei
2. Bob queues 1 deposit for 1e18

Since there's no way to force Alice to process her deposits, Bob can either wait for somebody else to process them or process them himself. However, whoever does that will have to pay enormous amount of gas fees.

Here's a PoC using Foundry, making use of `LiquidityPoolTest` (`LiquidityPool.Deposits.t.sol#L12`) test suite. However, it would work pretty much in the same way for `KangarooVault` , where the only prerequisite would be that there have to be currently active positions.

```
function testDepositFee() public {
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");
    susd.mint(alice, 1e18);
    susd.mint(bob, 1e18);

    vm.startPrank(alice);
    susd.approve(address(pool), 1e18);

    for(uint256 i = 0; i < 100000; i++) {
        pool.queueDeposit(1 wei, alice);
    }
    vm.stopPrank();

    vm.startPrank(bob);
    susd.approve(address(pool), 1e18);
    pool.queueDeposit(1e18, address(bob));

    vm.warp(block.timestamp + pool.minDepositDelay());

    pool.processDeposits(100001);

    vm.stopPrank();
}
```

To see the result yielded by running the test using the following command `forge t --match-test testDepositFee -vv --gas-report` , see the warden's **original submission**.

The same scenario can happen when processing withdrawals. If we expand the example to extreme values (100,000,000 deposits), this would mean that the approximate gas to be paid for users that want to withdraw their funds equal approximately 2448749420000, which converted to today's ETH:USD ratio is around 4070335.77 USD.

Tools Used

Manual review, foundry.

Recommended Mitigation Steps

My recommendations for this vulnerability are the following:

1. Replace the sequential processing of deposits and withdrawals with functionality where users can execute their own deposits and withdrawals without having to process all the deposits and withdrawals before theirs.
2. If you insist keeping the sequential processing, add a mechanism to cancel deposits but still implement the second part of point 1.

Dravee (judge) decreased severity to Medium and commented:

> Due to the high quality of the report and the details given (particularly, the real impact on the user's funds), I'll be selecting this for the report.
> While High Severity can be defended here, I believe this to be more of a Griefing attack (Medium Severity, no profit motive for an attacker, but damage done to the users or the protocol).
> The user's assets in the protocol aren't at direct risk, even if they need to pay more Gas (on Optimism).
> On Immunefi, "Theft of gas" or "Unbounded gas consumption" are considered Medium Severity issues, and some projects even put "Loss of gas costs or funds will not be considered 'loss of funds'" in their OOS section (like Olympus).
> Hence the Medium Severity.

mubaris (Polynomial) confirmed via duplicate issue `#122`

[M-11] The Liquidity Pool will lack margin of Synthetix perpetual market, if `liquidationBufferRatio` of contract `PerpsV2MarketSettings` from Synthetix is updated

Submitted by KIntern_NA

The Liquidity Pool can lack the margin of PerpMarket if `liquidationBufferRatio` of contract `PerpsV2MarketSettings` from Synthetix is greater than 1e18. Then the delay orders of the pool can not be executed and the position of the pool might be liquidated.

Proof of concept

- Function `_calculateMargin` returns the margin amount needed to transfer with the specific size of `PerpMarket`.

```
function _calculateMargin(int256 size) internal view returns (ui
    (uint256 spotPrice, bool isInvalid) = baseAssetPrice();

    require(!isInvalid || spotPrice > 0);

    uint256 absSize = size.abs();
    margin = absSize.mulDivDown(spotPrice, futuresLeverage);
}
```

- When `LiquidityPool` executes a delayed order of Synthetix, function `_updatePositionMargin` (`PerpsV2MarketProxyable.sol#L133`) from PerpsV2Market of Synthetix will be called. It will check the margin of the new position in the perps market:

```
uint liqPremium = _liquidationPremium(position.size, price);
uint liqMargin = _liquidationMargin(position.size, price).add(li
_revertIfError(
    (margin < _minInitialMargin()) ||
        (margin <= liqMargin) ||
        (_maxLeverage(_marketKey()) < _abs(_currentLeverage(posi
    Status.InsufficientMargin
);
```

- Function `_liquidationMargin` (`PerpsV2MarketBase.sol#L390`) returns the maximum of margin that will be liquidated with the position size of perps market. Then the new margin must to be greater than `_liquidationMargin`.

```
function _liquidationMargin(int positionSize, uint price) inter
    uint liquidationBuffer = _abs(positionSize).multiplyDecimal(
    return liquidationBuffer.add(_liquidationFee(positionSize, p
```

}

- To calculate the `_liquidationMargin`, `PerpsV2Martket` use the variable `_liquidationBufferRatio` (`MixinPerpsV2MarketSettings.sol#L171`) as the scale of `_abs(positionSize).multiplyDecimal(price)` (value of the position). This variable has getter and setter functions in the contract `PerpsV2MarketSettings` (`PerpsV2MarketSettings.sol`). You can find this contract at https://optimistic.etherscan.io/address/0x09793Aad1518B8d8CC72FDd356479E3CBa7B4Ad1#code.
- `_liquidationBufferRatio` is 1e16 (1%) now but can be changed in the future, and can become market-specific (I asked Synthetix team and they said it will be changed in a couple of weeks, but I didn't know how it will be changed).
- Since function `_calculateMargin` in contract LiquidityPool doesn't consider this minimum required margin (to not be liquidated), LiquidityPool can lack the margin of perps market in the future.
- Scenario:
- `futuresLeverage` of LiquidityPool is applied to be 5.Then function `_calculateMargin` returns 1/5 (20%) amount of position value (position value = size * spotPrice)
- `_liquidationBufferRatio` is set to be 3e17 (30%) in `PerpsV2MarketSettings` contract. Then it requires a margin >= 30% of the position value when updating a position.
- LiquidityPool's margin is not enough for its position in PerpsMarket. Then the delay orders of the pool can't be executed and the position of the pool in PerpsMarket can be liquidated

Recommended Mitigation Steps

Should calculate `_liquidationMargin` from PerpsMarket using the current `_liquidationBufferRatio` from `PerpsV2MarketSettings` contract, to set the minimum margin in function `_calculateMargin`.

mubaris (Polynomial) disputed and commented:

> Liquidation margin can't be above 1e18. `futuresLeverage` is under the control of the admin and we expect to keep it at respectable values like 2 where Synthetix provides 25x leverage and it is expected to set to 100x in the future.

[Dravee (judge) invalidated and commented](#):

> From the warden's submission above:

> *if liquidationBufferRatio of contract PerpsV2MarketSettings from Synthetix is greater than 1e18*

> *_liquidationBufferRatio is 1e16 (1%) now but can be changed in the future, and can become market-specific (I asked Synthetix team and they said it will be changed in a couple of weeks, but I didn't know how it will be changed).*

> ChatGPT to the rescue:

> *The liquidation buffer ratio is a metric used in cryptocurrency trading to determine the level of risk associated with holding a leveraged position. When trading with leverage, a trader borrows funds to increase their trading position, and the liquidation buffer ratio represents the amount of collateral a trader must hold to avoid being liquidated in the event of a market downturn.*

> *The liquidation buffer ratio is calculated by dividing the collateral held by the trader by the notional value of their leveraged position. For example, if a trader has `$10,000` worth of collateral and a leveraged position with a notional value of `$100,000`, their liquidation buffer ratio would be 10% (10,000 / 100,000).*

> *If the value of the trader's position falls below a certain threshold determined by the exchange, the trader's position will be automatically liquidated to repay the borrowed funds, which can result in significant losses. Maintaining a sufficient liquidation buffer ratio can help traders manage their risk and avoid liquidation.*

> Hence the starting hypothesis is indeed implausible.

[duc (warden) commented](#):

> I made a typing mistake in the impact assessment, where the number 1e18 should be 1e16 (1%). This is the current value of `liquidationBufferRatio` in Synthetix perps, although it may change in the future. In the proof of concept, I used 3e17 (30%) as an example, and it is a valid value for `_liquidationBufferRatio`.

mubaris (Polynomial) commented:

> Realistically, the protocol would be using 2-3x leverage (unlike 5x mentioned by the warden). Synthetix changing params takes at least a week and they announce it via SIPs. In that time, we can always reduce the leverage or add additional margin. Also anything above 10% liquidation buffer is absurd.

rivalq (Polynomial) confirmed and commented:

> Yeah this scenario can happen but only after many what-ifs.

Dravee (judge) marked as valid

[M-12] Attacker can post-running attack to prevent `LiquidityPool` from hedging by orders of PerpMarket

Submitted by KIntern_NA

The attacker can post-running attack to keep the `LiquidityPool's` can't submit the orders of perpetual for hedging. It leads to every trade of the pool will not be hedged anymore.

Proof of concept

1. `LiquidityPool` calls function `_hedge` every trade, and it triggers function `_placeDelayedOrder`.

```
function _placeDelayedOrder(int256 sizeDelta, bool isLiquidatior
    IPerpsV2MarketBaseTypes.DelayedOrder memory order = perpMark

    (,,,,, IPerpsV2MarketBaseTypes.Status status) =
        perpMarket.postTradeDetails(sizeDelta, 0, IPerpsV2Market
```

```
        int256 oldSize = order.sizeDelta;
        if (oldSize != 0 || isLiquidation || uint8(status) != 0) {
            queuedPerpSize += sizeDelta;
            return;
        }
        perpMarket.submitOffchainDelayedOrderWithTracking(sizeDelta,

        emit SubmitDelayedOrder(sizeDelta);
    }
```

2. In this function, if `(oldSize != 0 || isLiquidation || uint8(status) != 0)` , the pool will accumulate the variable `queuedPerpSize` and return. Else, the pool will submit a delay order of `sizeDelta` (the current size delta of the trade) to the Synthetix perpetual market.
3. The current delay order of Pool will be executed by function `executePerpOrders` (`LiquidityPool.sol#L704`). After that, the order size of the pool will return to 0 and the pool can submit the other delayed order.
4. However, when the pool can submit a new order, function `_placeDelayedOrder` just submit the order with `sizeDelta` of the current trade. And the order of `queuedPerpSize` can only submitted in function `placeQueuedOrder` (`LiquidityPool.sol#L692`), but it require the current order size of pool is 0:

```
    function placeQueuedOrder() external requiresAuth nonReentrant {
        IPerpsV2MarketBaseTypes.DelayedOrder memory order = perpMark

        require(order.sizeDelta == 0);
```

5. Therefore, after the `executePerpOrders` call from the authority, attacker can post-run opening/closing a position, to trigger function `_placeDelayedOrder` , and make the pool submit the order of the current `sizeDelta` . Then the order size of the pool will be different from 0, and the pool can't submit other delay orders, until the next `executePerpOrders` call. And all the sizes of trades after that will be accumulated into `queuedPerpSize` .
6. Attacker can repeat post-running function `executePerpOrders` with a small trade, to keep the pool can't submit the necessary order (with

`queuedPerpSize`) for hedging. Then every trade will not be hedged.

Recommended Mitigation Steps

Function `_placeDelayedOrder` should submit the order of `queuedPerpSize + sizeDelta` instead of `sizeDelta`. You can take a look at the following example:

```
function _placeDelayedOrder(int256 sizeDelta, bool isLiquidatior
    IPerpsV2MarketBaseTypes.DelayedOrder memory order = perpMark

    (,,,,, IPerpsV2MarketBaseTypes.Status status) =
        perpMarket.postTradeDetails(queuedPerpSize + sizeDelta,

    int256 oldSize = order.sizeDelta;
    if (oldSize != 0 || isLiquidation || uint8(status) != 0) {
        queuedPerpSize += sizeDelta;
        return;
    }
    perpMarket.submitOffchainDelayedOrderWithTracking(queuedPerp

    emit SubmitDelayedOrder(sizeDelta);
}
```

mubaris (Polynomial) acknowledged, but disagreed with severity and commented:

> There's a function to manually hedge the pool if it is not `hedgePositions()` , but I acknowledge the issue but disagree with the severity of this.

Dravee (judge) decreased severity to Medium and commented:

> Griefing attack that has a counter. I believe Medium to be right.

[M-13] Attacker can transfer all SUSD's balance of LiquidityPool as margin to Synthetix perpetual market, and break the actions of users until the pool is rebalanced by the authority

Submitted by KIntern_NA

Contract `LiquidityPool` will transfer margin of size delta for every trade, and this margin is always > 0. Then attacker can repeat open and close a position in 1 transaction, to make the pool transfer all its SUSD tokens to Synthetix perpetual market as margin, even the perpetual size of `LiquidityPool` will not change after that. Then many actions of users which need SUSD of the pool such as withdrawing liquidity tokens, opening short positions, closing long positions... can't be executed until the pool is rebalanced by the authority.

Proof of concept

Let's take a look at function `_hedge` in contract `LiquidityPool`:

```
function _hedge(int256 size, bool isLiquidation) internal returr
    ...
    uint256 marginRequired = _calculateMargin(hedgingSize) + hec
    usedFunds += int256(marginRequired);
    require(usedFunds <= 0 || totalFunds >= uint256(usedFunds));

    perpMarket.transferMargin(int256(marginRequired));
    ...
```

In this function, `marginRequired` is always > 0, since it is the required margin for the independant `hedgingSize`.

Even the size of pool's perpetual position increases or decreases (sometimes it doesn't need to transfer more positive margin), it always transfer this amount of SUSD as margin to Synthetix perpetual market.

Then attacker can make the pool transfer all its SUSD tokens as margin by repeating open and close a position, although it will not change the pool's perpetual size after that. It leads to users' actions can be broken because of the lack of SUSD in LiquidityPool, until the pool is rebalanced by the authority. Furthermore, the attacker can front-run to break the actions of important specific users.

Recommended Mitigation Steps

Calculate `marginRequired` (can be positive or negative) from the new perpetual size and the remaining margin. I advise you to use the similar calculation from the function `rebalanceMargin`.

mubaris (Polynomial) disagreed with severity and commented:

> The entire action of the pool will be watched by a keeper bot to call `rebalanceMargin()` anytime required. I disagree with the severity of this issue.

Dravee (judge) decreased severity to Low/Non-Critical and commented:

> *Furthermore, the attacker can front-run to break the actions of important specific users.*

> Optimism, no front-running.

> Assets aren't at risk and this can't really be considered a real grief attack either if this is just a random act.

> Lack of coded POC too to prove that this could actually be done for cheap or not by the attacker. Hand-waved arguments.

> Will downgrade to QA.

duc (warden) commented:

> This issue highlights the problem that the LiquidityPool contract always adds margin for every trade, even if sizeDelta is decreased. The marginRequired should be calculated correctly, similar to the rebalanceMargin function, to prevent the transferred margin from growing too high.

> Therefore, the attacker can conduct a grief attack by repeatedly opening and closing a position in 1 transaction, causing the LiquidityPool to transfer more margin than it actually needs. I am aware that a keeper bot will be used to call rebalanceMargin() whenever necessary, but bot's action can't guarantee flawless performance indefinitely. So within the scope of the smart contract, I think this issue deserves to be considered a valid medium.

Dravee (judge) increased severity to Medium and commented:

> Talked with the sponsor.
> This issue can indeed be considered valid, but will be a no-fix.
> Still a nice-to-have on the final report.

[M-14] Possible spamming attack in opening or closing Long or Short Positions in `Exchange.openTrade`

Submitted by PaludoX0, also found by Bauer

One user that wants to open a short position shall not have a long position already opened and viceversa.

This is the check for opening a long position:

```
        if (params.isLong) {
            uint256 shortPositions = shortToken.balanceOf(msg.sender
            require(shortPositions == 0, "Short position must be clo
```

This is the check for opening a short position:

```
        } else {
            uint256 holdings = powerPerp.balanceOf(msg.sender);
            require(holdings == 0, "Long position must be closed bef
```

The issue is that anyone can open a short position with value 0 without binding any collateral and then spamming the short token to user that would like to open a position. Neither in `Exchange._openTrade` or `ShortToken.sol.adjustPosition` there's a check the amount of short shall be > 0

Proof of Concept

Copy and paste the following POC in `test/Exchange.Simple.t.sol`

```
    function testMultipleShortOpen0AmountandSpam() public {
    //Open a short position for user_1 with 0 shortAmount and 0 coll
        for (uint256 i = 0; i < 1000; i++) {
```

```
        (uint256 positionId, Exchange.TradeParams memory tradePa
        (uint256 _positionId, uint256 _shortAmount, uint256 _col

    //(, uint256 _collateralAmount) = shortCollateral.userCollat
        assertEq(_shortAmount, 0);
        assertEq(positionId, i);
        assertEq(shortToken.balanceOf(user_1), i);
        assertEq(_collateralAmount, 0);
    }

//Open Long for user_1 and it will revert because it has at leas
    susd.mint(user_1, 1000e18);
    Exchange.TradeParams memory tradeParamsL;
    tradeParamsL.isLong = true;
    tradeParamsL.amount = 1e18;
    tradeParamsL.maxCost = 1000e18;
vm.startPrank(user_1);
    susd.approve(address(getPool()), 1000e18);

vm.expectRevert();
    exchange.openTrade(tradeParamsL);

//Perp balance of user_1 is 0
assertEq(powerPerp.balanceOf(user_1), 0);

//user_1 transfers a shortToken to user_2
    uint256 positionId =55;
    shortToken.safeTransferFrom(user_1, user_2, positionId);
vm.stopPrank();  //end of domain of user_1

//Open Long for user_2 it will revert because it has a short ope

    susd.mint(user_2, 1000e18);
    tradeParamsL.isLong = true;
    tradeParamsL.amount = 2e18;
    tradeParamsL.maxCost = 1000e18;
    vm.startPrank(user_2);
    susd.approve(address(getPool()), 1000e18);
vm.expectRevert();
    exchange.openTrade(tradeParamsL);
vm.stopPrank();
assertEq(powerPerp.balanceOf(user_2),0);
}
```

Tools Used

Manual review and foundry forge

Recommended Mitigation Steps

It’s suggested to set a sensible minimum amount of tokens to be withdrawn or at least to be greater than 0.

Dravee (judge) commented:

> The POC fails, but can be corrected with the following:

```
-    for (uint256 i = 0; i < 1000; i++) {
+    for (uint256 i = 1; i < 1000; i++) {
```

Dravee (judge) decreased severity to Medium

mubaris (Polynomial) confirmed

[M-15] First Depositors will incurr a rebalancing Loss

Submitted by GalloDaSballo

```
    function getTokenPrice() public view returns (uint256) {
        if (totalFunds == 0) {
            return 1e18;
        }

        uint256 totalSupply = getTotalSupply();
        if (positionData.positionId == 0) {
            return totalFunds.divWadDown(totalSupply);
        }
```

After a deposit, the funds are invested, doing so incurs a fee, which causes `getTokenPrice` to reduce, this offers a discount to later depositors.

Because the fee impacts the token price, future depositors will be able to purchase the Vault token at a discount.

This gives them an advantage as they'll be receiving yield without incurring the additional cost.

Coded POC

The following POC is built by creating a new test file `TestRebaseAttack.t.t.sol`

The salient output from the test is the following

```
├─ emit Debug(name: tokenPrice, value: 1000000000000000000)
├─ emit Debug(name: newTokenPrice, value: 99868400000000000(
```

Which means that after creating the position to be delta neutral, due to fees being paid, the price of the token is lower, meaning it's cheaper to deposit after `openPosition` instead of before.

Below the full code for you to verify:

Can be run with `forge test --match-test testOpenRebaseAttack -vvvvv`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

import {console2} from "forge-std/console2.sol";

import {FixedPointMathLib} from "solmate/utils/FixedPointMathLik
import {Exchange} from "../src/Exchange.sol";
import {TestSystem} from "./utils/TestSystem.sol";
import {KangarooVault} from "../src/KangarooVault.sol";
import {MockERC20Fail} from "../src/test-helpers/MockERC20Fail.s
import {LiquidityPool} from "../src/LiquidityPool.sol";
import {IPerpsV2MarketBaseTypes} from "../src/interfaces/synthet

contract TestRebaseAttack is TestSystem {
    using FixedPointMathLib for uint256;

    uint256 public constant initialPrice = 1200e18;
```

```
Exchange private exchange;
MockERC20Fail private susd;
KangarooVault private kangaroo;
LiquidityPool private pool;

uint256 private leverage;
uint256 private collRatio;

bytes[] private emptyData;

function setUp() public {
    deployTestSystem();
    initPool();
    initExchange();
    preparePool();
    setAssetPrice(initialPrice);
    initKangaroo();

    exchange = getExchange();
    susd = getSUSD();
    kangaroo = getKangaroo();
    pool = getPool();

    susd.mint(user_1, 5e23);

    vm.startPrank(user_1);
    susd.approve(address(kangaroo), 5e23);
    kangaroo.initiateDeposit(user_1, 1e23);
    vm.stopPrank();

    leverage = kangaroo.leverage();
    collRatio = kangaroo.collRatio();
}


event Debug(string name, uint256 value);

function testOpenRebaseAttack() public {
    uint256 tokenPrice = kangaroo.getTokenPrice();
    assertEq(tokenPrice, 1e18, "!1e18");

    kangaroo.openPosition(1e19, 0);

    uint256 newTokenPrice = kangaroo.getTokenPrice();
```

```
        vm.startPrank(user_1);
        kangaroo.initiateDeposit(user_1, 1e18);
        vm.stopPrank();

        emit Debug("tokenPrice", tokenPrice);
        emit Debug("newTokenPrice", newTokenPrice);

        assertTrue(tokenPrice != newTokenPrice, "No change");
    }
}
```

Recommended Mitigation

It may be best to have a deposit or withdrawal fee that evens out the hedging costs as not to discourage early depositors.

mubaris (Polynomial) disputed and commented:

> Token price is meant to go down after you open a position, that's how it's supposed to work.

Dravee (judge) commented:

> *Token price is meant to go down after you open a position, that's how it's supposed to work.*

> Very hard to argue with such an argument. If the warden can think of some counter-arguments in Post-Judging QA, I'll be open to hear them.

GalloDaSballo (warden) commented:

> Thank you for the opportunity to add more info. I believe the initial submission shows that the price for consecutive depositors is cheaper. For example if we let a big deposit happen, then we will get more tokens for less cost.

> The updated tests simply has a second depositor instead of checking the price, you can see that if I deposit as the second person I get 5 times more tokens.

```
function testOpenRebaseAttack() public {
```

```
        uint256 tokenPrice = kangaroo.getTokenPrice();
        assertEq(tokenPrice, 1e18, "!1e18");

        kangaroo.openPosition(1e19, 0);

        uint256 newTokenPrice = kangaroo.getTokenPrice();

        vm.startPrank(user_2);
        kangaroo.initiateDeposit(user_2, 5e23);
        vm.stopPrank();

        kangaroo.processDepositQueue(1);


        emit Debug("tokenPrice", tokenPrice);
        emit Debug("newTokenPrice", newTokenPrice);
        emit Debug("VAULT_TOKEN.balanceOf(user_2)", VAULT_TOKEN.
        emit Debug("VAULT_TOKEN.balanceOf(user_1))", VAULT_TOKEN

        assertTrue(tokenPrice != newTokenPrice, "No change");
        assertTrue(VAULT_TOKEN.balanceOf(user_2) > VAULT_TOKEN.b
    }
```

> Log shows that I get 5e23 tokens vs 1e23

```
    ├─ emit Debug(name: tokenPrice, value: 1000000000000000000)
    ├─ emit Debug(name: newTokenPrice, value: 99868400000000000(
    ├─ [542] VaultToken::balanceOf(0x53E2Cd188FE5E37D9bA9D26782{
    │   └─ ← 500658867069062886758974
    ├─ emit Debug(name: VAULT_TOKEN.balanceOf(user_2), value: 5(
    ├─ [2542] VaultToken::balanceOf(0xe5ecc448cf264e5AB26D08C16e
    │   └─ ← 100000000000000000000000
    ├─ emit Debug(name: VAULT_TOKEN.balanceOf(user_1)), value: 1
    ├─ [542] VaultToken::balanceOf(0xe5ecc448cf264e5AB26D08C16e3
    │   └─ ← 100000000000000000000000
    ├─ [542] VaultToken::balanceOf(0x53E2Cd188FE5E37D9bA9D26782{
    │   └─ ← 500658867069062886758974
    └─ ← ()
```

[mubaris (Polynomial) acknowledged and commented]:

> What you're saying is true, but token price reflects the price of each token against the total asset held by the vault. Once you pay fee for an action like trading, that

> fee is lost forever from the vault. If we start accounting that (some value that doesn't exist), we'll run in to accounting errors down the line. So it doesn't make sense to add the fees paid by the vault in value held by the vault.

> It is also true that, the second user gets much more tokens than the first user. But any funds added to the vault gets used for opening a new position and the user has stay in the vault until the new position becomes profitable to be profitable. There's no immediate exit process here. We use the same mechanism for our options vaults which has been live for a while now.

[M-16] Supply drain of PowerPerp tokens through liquidations

Submitted by [RaymondFam]

Contract `PowerPerp` that has Solmate's ERC20.sol inherited has zero `totalSupply` initiated by default. And, as denoted by PowerPerp.sol, `onlyExchange` can mint or burn Power (Square) Perp ERC-20 tokens:

File: `PowerPerp.sol#L32-L38`

```
    function mint(address _user, uint256 _amt) public onlyExchar
        _mint(_user, _amt);
    }

    function burn(address _user, uint256 _amt) public onlyExchar
        _burn(_user, _amt);
    }
```

This could run into supply issue if the source and drain have not been routed through the right channels.

Proof of Concept

There is only one source of minting when users open long trades (`Exchange.sol#L233-L245`).

However, there are two channels users could burn the tokens, i.e. closing long trades (`Exchange.sol#L288-L299`), and liquidating positions (`Exchange.sol#L333-L353`).

It is apparent that the long traders hold the majority of the tokens although long term the tokens can be swapped for `SUSD` on the swap exchange, believing that is how `MarkPrice` derives from. Additionally, this should be where liquidators who are neither long nor short traders buy their Power Perp tokens from prior to profiting on the liquidable positions.

If liquidations get more frequent than expected due to collateral token dropping in price, `powerPerp.burn()` (`Exchange.sol#L350`) is going to both strain the Power Perp token supply and drive up the price. As a result, the amount of positions going underwater is going to increase too.

The chain effect continues since long traders will have difficulty closing their positions if their minted Power Perp tokens have earlier been swapped for `SUSD` at lower prices than now.

The situation could be worse if the scenario described above were to happen in the early phase of system launch.

Recommended Mitigation Steps

Since short traders are sent the cost deducted `SUSD` when opening their positions, consider having liquidators sending in the equivalent amount of cost added `SUSD` to the liquidity pool (just like when short traders are closing their positions) instead of having Power Perp tokens burned in `_liquidate()`. This will also have a good side effect of enhancing the hedging capability of the liquidity pool.

[mubaris (Polynomial) confirmed]

[M-17] Users' collateral could get stuck permanently after fully closing short trades

Submitted by [RaymondFam]

When completely closing a short trade, a user is supposed to input `TradeParams` such that:

`shortPosition.shortAmount == params.shortAmount`

`shortPosition.collateralAmount == params.collateralAmount`

However, if cares have not been given particularly when inputting `params.collateralAmount` via a non-frontend method such as https://optimistic.etherscan.io/, a zero or a value smaller than `shortPosition.collateralAmount` could be accidentally entered. After the transaction has succeeded, the user's collateral would be permanently locked in contract `ShortCollateral`.

Proof of Concept

As can be seen from the code block pertaining to `_closeTrade()` below, `totalShortAmount == 0` will make the require statement pass easily because `minCollateral == 0` (`ShortCollateral.sol#L169-L170`).

File: `Exchange.sol#L310-L319`

```
            uint256 totalShortAmount = shortPosition.shortAmount
            uint256 totalCollateralAmount = shortPosition.collat

            uint256 minCollateral = shortCollateral.getMinCollat
            require(totalCollateralAmount >= minCollateral, "Not

            shortCollateral.sendCollateral(params.positionId, pa
            shortToken.adjustPosition(
                params.positionId, msg.sender, params.collateral
            );
```

The inadequate `params.collateralAmount` accidentally inputted is then sent to the user:

File: `ShortCollateral.sol#L106-L116`

```
    function sendCollateral(uint256 positionId, uint256 amount)
        UserCollateral storage userCollateral = userCollaterals[

        userCollateral.amount -= amount;
```

```
        address user = shortToken.ownerOf(positionId);

        ERC20(userCollateral.collateral).safeTransfer(user, amou

        emit SendCollateral(positionId, userCollateral.collatera
    }
```

Next, the user's position is adjusted such that its position is burned because of a complete position close. Note that `position.shortAmount` is assigned `0` whereas `position.collateralAmount` is assigned a non-zero value.

File: `ShortToken.sol#L79-L84`

```
            position.collateralAmount = collateralAmount;
            position.shortAmount = shortAmount;

            if (position.shortAmount == 0) {
                _burn(positionId);
            }
```

Because the user's ERC-721 short token is now burned, removing the forgotten/remaining collateral from the short position is going to revert on the ownership check:

File: `Exchange.sol#L388`

```
        require(shortToken.ownerOf(positionId) == msg.sender);
```

Recommended Mitigation Steps

Consider adding a check in `_closeTrade()` that will fully send the collateral to the user when the position is intended to be fully closed as follows:

File: `Exchange.sol#L310-L316`

```
            uint256 totalShortAmount = shortPosition.shortAmount
```

```
            uint256 totalCollateralAmount = shortPosition.collat

            uint256 minCollateral = shortCollateral.getMinCollat
            require(totalCollateralAmount >= minCollateral, "Not

+             if (totalShortAmount == 0) {
+                 params.collateralAmount = shortPosition.collate
+             }
            shortCollateral.sendCollateral(params.positionId, pa
```

[mubaris (Polynomial) confirmed]()

[M-18] Lack of price validity check from Synthetix results in loss of funds while liquidating

Submitted by [DadeKuma](), *also found by* [rbserver]() *and* [__141345__]()

Lack of a validity check while liquidating results in loss of funds, as the price could be invalid momentarily from Synthetix due to high volatility or other issues.

Proof of Concept

There isn't a check for `isInvalid` in `getMarkPrice` and `getAssetPrice`, which must be false before closing the liquidation:

```
File: src/ShortCollateral.sol

134:         (uint256 markPrice,) = exchange.getMarkPrice();
135:         (uint256 collateralPrice,) = synthetixAdapter.getAs
```

This check is used in other similar functions that fetch the price:

```
File: src/ShortCollateral.sol

194:         (uint256 markPrice, bool isInvalid) = exchange.getM
195:         require(!isInvalid);

205:         (collateralPrice, isInvalid) = synthetixAdapter.get
```

```
206:          require(!isInvalid);
```

This must be present to ensure that the price fetched from Synthetix is not stale or invalid.

If this isn't the case, a liquidation could result in under-liquidation (a loss for the user) or over-liquidation (a loss for the protocol).

The same problem is also present in `LiquidityPool`:

```
File: src/LiquidityPool.sol

388:          (uint256 markPrice,) = exchange.getMarkPrice();
```

As the `markPrice` is not validated when calculating the `orderFee`.

Recommended Mitigation Steps

Add a check to be sure that `isInvalid` is false in both `markPrice` and `collateralPrice` before liquidating.

rivalq (Polynomial) disagreed with severity

Dravee (judge) commented:

> Would like @rivalq 's thought on the severity and validity.
> Was there a reason for an absence on these checks? (Like a redundancy because it would revert somewhere on an invalid price).

> It was also raised by the warden that an invalid price could be 0 through these:

- `PerpsV2MarketViews.sol#L45-L50`
- `PerpsV2Market.sol#L124-L126`

> How likely is this to happen?

mubaris (Polynomial) confirmed and commented:

> This seems like a miss from our side.

Dravee (judge) commented:

> Agreed on Medium severity.

[M-19] Collateral removal not possible

Submitted by csanuragjain, also found by DadeKuma, rbserver, bytes032, and KIntern_NA

If an approved collateral has later started say taking fees on transfer then protocol has no way to remove such collateral. The current deposit logic cannot handle fee on transfer token and would give more funds to user then actually obtained by contract

Proof of Concept

1. Assume protocol was supporting collateral X (say USDT which has fee currently set as 0)
2. After some time collateral introduces fee on transfer
3. Protocol does not have a way to remove a whitelisted collateral
4. Problem begins once user starts depositing such collateral

```
function _addCollateral(uint256 positionId, uint256 amount)
...
ERC20(shortPosition.collateral).safeTransferFrom(msg.sender,
        ERC20(shortPosition.collateral).safeApprove(address(

        shortToken.adjustPosition(
            positionId,
            msg.sender,
            shortPosition.collateral,
            shortPosition.shortAmount,
            shortPosition.collateralAmount + amount
        );
        shortCollateral.collectCollateral(shortPosition.coll
...
```

```
        }
```

5. In this case `amount` is transferred from user to contract but contract will only receive `amount-fees`. But contract will still adjust position with full `amount` instead of `amount-fees` which is incorrect.

Recommended Mitigation Steps

Add a way to disapprove collateral so that if in future some policy changes for a particular collateral, protocol can stop supporting it. This will it would only have to deal with existing collateral which can be wiped out slowly using public announcement.

[Dravee (judge) commented]:

> Not a duplicate of [https://github.com/code-423n4/2023-03-polynomial-findings/issues/178] as Fee-on-transfer tokens are only mentioned as a scenario that may make the protocol want to disapprove a collateral.

> Due to a real lack of way to disapprove a collateral, I believe this finding is valid.

[mubaris (Polynomial) confirmed]

Low Risk and Non-Critical Issues

For this audit, 14 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below] by **rbserver** received the top score from the judge.

The following wardens also submitted reports: auditor0517, adriro, joestakey, btk, bin2chen, juancito, RaymondFam, PaludoX0, DadeKuma, 0xSmartContract, GalloDaSballo, Sathish9098, *and* Rolezn.

Summary

	Issue
[01]	LACK OF LIMITS FOR SETTING FEES
[02]	SOME FUNCTIONS DO NOT FOLLOW CHECKS-EFFECTS-INTERACTIONS PATTERN
[03]	`nonReentrant` MODIFIER CAN BE PLACED AND EXECUTED BEFORE OTHER MODIFIERS IN FUNCTIONS
[04]	REDUNDANT `return` KEYWORDS IN `ShortToken.transferFrom` and `ShortToken.safeTransferFrom` FUNCTIONS
[05]	CONSTANTS CAN BE USED INSTEAD OF MAGIC NUMBERS
[06]	HARDCODED STRING THAT IS REPEATEDLY USED CAN BE REPLACED WITH A CONSTANT
[07]	`ShortToken.adjustPosition` FUNCTION DOES NOT NEED TO UPDATE `totalShorts` and `position.shortAmount` IN CERTAIN CASE
[08]	`LiquidityPool.withdraw` FUNCTION CALLS BOTH `SUSD.safeTransfer` AND `SUSD.transfer`
[09]	`LiquidityPool.orderFee` FUNCTION CAN CALL `getMarkPrice()` INSTEAD OF `exchange.getMarkPrice()`
[10]	IMMUTABLES CAN BE NAMED USING SAME CONVENTION
[11]	UNUSED IMPORT
[12]	FLOATING PRAGMAS
[13]	SOLIDITY VERSION `0.8.19` CAN BE USED
[14]	ORDERS OF LAYOUT DO NOT FOLLOW OFFICIAL STYLE GUIDE
[15]	INCOMPLETE NATSPEC COMMENTS
[16]	MISSING NATSPEC COMMENTS

[01] LACK OF LIMITS FOR SETTING FEES

When calling the following `LiquidityPool.setFees` function, there are no limits for setting `depositFee` and `withdrawalFee`. If these fees are set to 1e18, calling the `LiquidityPool.deposit` function can cause all of the `amount` to become the deposit fees and zero liquidity tokens to be minted to the user, and calling the `LiquidityPool.withdraw` function can cause all of the `susdToReturn` to become the withdrawal fees and zero `SUSD` tokens to be transferred to the user. If these fees are set to be more than 1e18, calling the `LiquidityPool.deposit` function can revert because `amount - fees` underflows, and calling the `LiquidityPool.withdraw` function can also revert because `susdToReturn - fees` underflows.

```
function setFees(uint256 _depositFee, uint256 _withdrawalFee
    emit UpdateFees(depositFee, _depositFee, withdrawalFee,
    depositFee = _depositFee;
    withdrawalFee = _withdrawalFee;
}
```

```
function deposit(uint256 amount, address user) external over
    uint256 tokenPrice = getTokenPrice();
    uint256 fees = amount.mulWadDown(depositFee);
    uint256 amountForTokens = amount - fees;
    uint256 tokensToMint = amountForTokens.divWadDown(tokenP
    liquidityToken.mint(user, tokensToMint);
    totalFunds += amountForTokens;
    SUSD.safeTransferFrom(msg.sender, feeReceipient, fees);
    SUSD.safeTransferFrom(msg.sender, address(this), amountF
    ...
}
```

```
function withdraw(uint256 tokens, address user) external ove
    ...
    uint256 tokenPrice = getTokenPrice();
    uint256 susdToReturn = tokens.mulWadDown(tokenPrice);
    uint256 fees = susdToReturn.mulWadDown(withdrawalFee);
    SUSD.safeTransfer(feeReceipient, fees);
    SUSD.transfer(user, susdToReturn - fees);
    totalFunds -= susdToReturn;
    liquidityToken.burn(msg.sender, tokens);
    ...
}
```

Moreover, the `LiquidityPool.setDevFee` function has no limit for setting `devFee`, and similar issues can occur. For example, if `devFee` is set to more than 1e18, calling the `LiquidityPool.openLong` function will revert because `externalFee` is more than `feesCollected` and executing `feesCollected - externalFee` reverts.

As a mitigation, to prevent the `LiquidityPool.deposit`, `LiquidityPool.withdraw`, and `LiquidityPool.openLong` functions from behaving unexpectedly, the `LiquidityPool.setFees` and `LiquidityPool.setDevFee` functions can be updated to only allow the corresponding fees to be set to values that cannot exceed certain limits, which are reasonable values that are less than 1e18.

```
    function setDevFee(uint256 _devFee) external requiresAuth {
        emit UpdateDevFee(devFee, _devFee);
        devFee = _devFee;
    }
```

```
    function openLong(uint256 amount, address user, bytes32 refe
        external
        override
        onlyExchange
        nonReentrant
        returns (uint256 totalCost)
    {
        ...
        uint256 tradeCost = amount.mulWadDown(markPrice);
        uint256 fees = orderFee(int256(amount));
        totalCost = tradeCost + fees;

        SUSD.safeTransferFrom(user, address(this), totalCost);

        uint256 hedgingFees = _hedge(int256(amount), false);
        uint256 feesCollected = fees - hedgingFees;
        uint256 externalFee = feesCollected.mulWadDown(devFee);

        SUSD.safeTransfer(feeReceipient, externalFee);

        usedFunds -= int256(tradeCost);
        totalFunds += feesCollected - externalFee;
        ...
```

```
}
```

[02] SOME FUNCTIONS DO NOT FOLLOW CHECKS-EFFECTS-INTERACTIONS PATTERN

Functions like `LiquidityPool.withdraw` and `ShortCollateral.collectCollateral` below transfer the corresponding tokens before updating the relevant states, which do not follow the checks-effects-interactions pattern. In contrast, functions like `LiquidityPool.deposit` and `ShortCollateral.sendCollateral` below transfer the corresponding tokens after updating the relevant states. To reduce the potential attack surface and increase the level of security, please consider updating the functions that do not follow the checks-effects-interactions pattern to follow such pattern.

```
function withdraw(uint256 tokens, address user) external ove
    ...
    SUSD.safeTransfer(feeReceipient, fees);
    SUSD.transfer(user, susdToReturn - fees);
    totalFunds -= susdToReturn;
    liquidityToken.burn(msg.sender, tokens);
    ...
}
```

```
function collectCollateral(address collateral, uint256 posit
    external
    onlyExchange
    nonReentrant
{
    ERC20(collateral).safeTransferFrom(address(exchange), ac

    UserCollateral storage userCollateral = userCollaterals[

    if (userCollateral.collateral == address(0x0)) {
        userCollateral.collateral = collateral;
    }

    userCollateral.amount += amount;
    ...
}
```

```
    function deposit(uint256 amount, address user) external over
        uint256 tokenPrice = getTokenPrice();
        uint256 fees = amount.mulWadDown(depositFee);
        uint256 amountForTokens = amount - fees;
        uint256 tokensToMint = amountForTokens.divWadDown(tokenP
        liquidityToken.mint(user, tokensToMint);
        totalFunds += amountForTokens;
        SUSD.safeTransferFrom(msg.sender, feeReceipient, fees);
        SUSD.safeTransferFrom(msg.sender, address(this), amountF

        emit Deposit(user, amount, fees, tokensToMint);
    }
```

```
    function sendCollateral(uint256 positionId, uint256 amount)
        UserCollateral storage userCollateral = userCollaterals[

        userCollateral.amount -= amount;

        address user = shortToken.ownerOf(positionId);

        ERC20(userCollateral.collateral).safeTransfer(user, amou

        emit SendCollateral(positionId, userCollateral.collatera
    }
```

[03] `nonReentrant` MODIFIER CAN BE PLACED AND EXECUTED BEFORE OTHER MODIFIERS IN FUNCTIONS

As a best practice, the `nonReentrant` modifier could be placed and executed before other modifiers in functions to prevent reentrancies through other modifiers and make code more efficient. To follow the best practice, please consider placing the `nonReentrant` modifier before the `requiresAuth` modifier in the following functions.

```
src\KangarooVault.sol
  376: function openPosition(uint256 amt, uint256 minCost) exter
  383: function closePosition(uint256 amt, uint256 maxCost) exte
  389: function clearPendingOpenOrders(uint256 maxCost) external
  395: function clearPendingCloseOrders(uint256 minCost) externa
  401: function transferPerpMargin(int256 marginDelta) external
```

```
424: function addCollateral(uint256 additionalCollateral) exte
436: function removeCollateral(uint256 collateralToRemove) ext
450: function executePerpOrders(bytes[] calldata priceUpdateDa
```

[04] REDUNDANT `return` KEYWORDS IN `ShortToken.transferFrom` and `ShortToken.safeTransferFrom` FUNCTIONS

The following `ShortToken.transferFrom` and `ShortToken.safeTransferFrom` functions do not have `returns` but have `return` statements. Moreover, Solmate's corresponding `ERC721.transferFrom` and `ERC721.safeTransferFrom` functions do not return anything. Thus, these `ShortToken.transferFrom` and `ShortToken.safeTransferFrom` functions' `return` keywords are redundant. To improve the code quality, please consider removing the `return` keywords from these functions.

```
function transferFrom(address _from, address _to, uint256 _i
    require(powerPerp.balanceOf(_to) == 0, "Receiver has lon
    return ERC721.transferFrom(_from, _to, _id);
}

function safeTransferFrom(address _from, address _to, uint25
    require(powerPerp.balanceOf(_to) == 0, "Receiver has lon
    return ERC721.safeTransferFrom(_from, _to, _id);
}

function safeTransferFrom(address _from, address _to, uint25
    require(powerPerp.balanceOf(_to) == 0, "Receiver has lon
    return ERC721.safeTransferFrom(_from, _to, _id, data);
}
```

[05] CONSTANTS CAN BE USED INSTEAD OF MAGIC NUMBERS

To improve readability and maintainability, a constant can be used instead of the magic number. Please consider replacing the magic numbers, such as `1e18`, used in the following code with constants.

```
src\Exchange.sol
  191: fundingRate = fundingRate / 1 days;
  197: int256 normalizationUpdate = 1e18 - totalFunding;

src\ShortCollateral.sol
  235: if (safetyRatio > 1e18) return maxDebt;
  237: maxDebt = position.shortAmount / 2;
```

[06] HARDCODED STRING THAT IS REPEATEDLY USED CAN BE REPLACED WITH A CONSTANT

`sUSD` is repeatedly used in the `ShortCollateral` contract. For better maintainability, please consider replacing it with a constant.

```
    constructor(uint256 susdRatio, uint256 susdLiqRatio, uint256
        Auth(msg.sender, Authority(address(0x0)))
    {
        ...
        Collateral storage susd = collaterals["sUSD"];
        susd.currencyKey = "sUSD";
        ...
    }

    function refresh() public {
        ...
        Collateral storage susd = collaterals["sUSD"];
        susd.synth = synthetixAdapter.getSynth("sUSD");
    }
```

[07] `ShortToken.adjustPosition` FUNCTION DOES NOT NEED TO UPDATE `totalShorts` and `position.shortAmount` IN CERTAIN CASE

When calling the following `ShortToken.adjustPosition` function, if `positionId == 0` is false and `shortAmount` equals `position.shortAmount`, `totalShorts` and `position.shortAmount` will be unchanged. Hence, to increase the code's efficiency, this function can be updated to not update `totalShorts` and `position.shortAmount` in this case.

```
    function adjustPosition(
        uint256 positionId,
        address trader,
        address collateral,
        uint256 shortAmount,
        uint256 collateralAmount
    ) external onlyExchange returns (uint256) {
        if (positionId == 0) {
            ...
        } else {
            require(trader == ownerOf(positionId));

            ShortPosition storage position = shortPositions[posi

            if (shortAmount >= position.shortAmount) {
                totalShorts += shortAmount - position.shortAmour
            } else {
                totalShorts -= position.shortAmount - shortAmour
            }

            position.collateralAmount = collateralAmount;
            position.shortAmount = shortAmount;

            if (position.shortAmount == 0) {
                _burn(positionId);
            }
        }
        ...
    }
```

[08] `LiquidityPool.withdraw` FUNCTION CALLS BOTH `SUSD.safeTransfer` AND `SUSD.transfer`

The `LiquidityPool.withdraw` function calls `SUSD.safeTransfer(feeReceipient, fees)` and `SUSD.transfer(user, susdToReturn - fees)`. For consistency and a higher level of security, the `LiquidityPool.withdraw` function can be updated to call `SUSD.safeTransfer(user, susdToReturn - fees)` instead of `SUSD.transfer(user, susdToReturn - fees)`.

```
function withdraw(uint256 tokens, address user) external ove
    ...
    SUSD.safeTransfer(feeReceipient, fees);
    SUSD.transfer(user, susdToReturn - fees);
    ...
}
```

[09] `LiquidityPool.orderFee` FUNCTION CAN CALL `getMarkPrice()` INSTEAD OF `exchange.getMarkPrice()`

The following `LiquidityPool.orderFee` function calls `exchange.getMarkPrice()` while all other functions in the same contract that need to call `exchange.getMarkPrice()`, such as the `LiquidityPool.getTokenPrice` function below, call `getMarkPrice()` instead. To make code more consistent and better, please consider updating the `LiquidityPool.orderFee` function to call `getMarkPrice()` instead of `exchange.getMarkPrice()`.

```
function orderFee(int256 sizeDelta) public view override ret
    ...
    (uint256 markPrice,) = exchange.getMarkPrice();
    ...
}
```

```
function getTokenPrice() public view override returns (uint2
    ...
    (uint256 markPrice, bool isInvalid) = getMarkPrice();
    ...
}
```

```
function getMarkPrice() public view override returns (uint25
    return exchange.getMarkPrice();
}
```

[10] IMMUTABLES CAN BE NAMED USING SAME CONVENTION

As shown below, some immutables are named using capital letters and underscores while some are not. For a better code quality, please consider naming these immutables using the same naming convention.

```
src\Exchange.sol
  34: uint256 public immutable override PRICING_CONSTANT;

src\KangarooVault.sol
  60: bytes32 public immutable name;
  63: bytes32 public immutable UNDERLYING_SYNTH_KEY;
  66: ERC20 public immutable SUSD;
  69: IVaultToken public immutable VAULT_TOKEN;
  72: IExchange public immutable EXCHANGE;
  75: ILiquidityPool public immutable LIQUIDITY_POOL;
  78: IPerpsV2Market public immutable PERP_MARKET;

src\LiquidityPool.sol
  56: bytes32 public immutable baseAsset;
  65: ERC20 public immutable SUSD;

src\LiquidityToken.sol
  8: bytes32 public immutable marketKey;
  10: ISystemManager public immutable systemManager;

src\PowerPerp.sol
  9: bytes32 public immutable marketKey;
  10: ISystemManager public immutable systemManager;

src\ShortCollateral.sol
  46: ISystemManager public immutable systemManager;

src\ShortToken.sol
  9: bytes32 public immutable marketKey;
  11: ISystemManager public immutable systemManager;

src\SystemManager.sol
  21: bytes32 public immutable baseAsset;
  24: ERC20 public immutable SUSD;
  27: bytes32 public immutable PERP_MARKET_CONTRACT;
```

[11] UNUSED IMPORT

The `IFuturesMarket` interface is not used in the `SystemManager` contract. Please consider removing the corresponding `import` statement for better readability and maintainability.

```
import {IFuturesMarket} from "./interfaces/synthetix/IFuturesMar
...
contract SystemManager is ISystemManager, Auth {
```

[12] FLOATING PRAGMAS

It is a best practice to lock pragmas instead of using floating pragmas to ensure that contracts are tested and deployed with the intended compiler version. Accidentally deploying contracts with different compiler versions can lead to unexpected risks and undiscovered bugs. Please consider locking pragmas for the following files.

```
src\Exchange.sol
  2: pragma solidity ^0.8.9;

src\KangarooVault.sol
  2: pragma solidity ^0.8.9;

src\LiquidityPool.sol
  2: pragma solidity ^0.8.9;

src\LiquidityToken.sol
  2: pragma solidity ^0.8.9;

src\PowerPerp.sol
  2: pragma solidity ^0.8.9;

src\ShortCollateral.sol
  2: pragma solidity ^0.8.9;

src\ShortToken.sol
  2: pragma solidity ^0.8.9;

src\SynthetixAdapter.sol
  2: pragma solidity ^0.8.9;

src\SystemManager.sol
  2: pragma solidity ^0.8.9;
```

```
src\libraries\SignedMath.sol
  2: pragma solidity ^0.8.9;

src\utils\PauseModifier.sol
  3: pragma solidity ^0.8.9;
```

[13] SOLIDITY VERSION `0.8.19` CAN BE USED

Using the more updated version of Solidity can enhance security. As described in https://github.com/ethereum/solidity/releases, Version `0.8.19` is the latest version of Solidity, which “contains a fix for a long-standing bug that can result in code that is only used in creation code to also be included in runtime bytecode”. To be more secured and more future-proofed, please consider using Version `0.8.19` for all contracts, including the `VaultToken` contract that uses Version `0.8.9` currently.

```
pragma solidity 0.8.9;
```

[14] ORDERS OF LAYOUT DO NOT FOLLOW OFFICIAL STYLE GUIDE

https://docs.soliditylang.org/en/v0.8.19/style-guide.html#order-of-layout suggests that the following order should be used in a contract:

1. Type declarations
2. State variables
3. Events
4. Errors
5. Modifiers
6. Functions

Events or error are placed after functions in the following contracts. To follow the official style guide, please consider placing these events or error before all functions in these contracts.

```
src\ShortCollateral.sol
  16: contract ShortCollateral is IShortCollateral, Auth, Reentr

src\ShortToken.sol
  8: contract ShortToken is ERC721 {

src\SystemManager.sol
  19: contract SystemManager is ISystemManager, Auth {

src\VaultToken.sol
  6: contract VaultToken is ERC20 {
```

[15] INCOMPLETE NATSPEC COMMENTS

NatSpec comments provide rich code documentation. The following functions miss the `@param` and/or `@return` comments. Please consider completing the NatSpec comments for these functions.

```
src\Exchange.sol
  87: function openTrade(TradeParams memory tradeParams)
  155: function getIndexPrice() public view override returns (ui
  186: function getMarkPrice() public view override returns (uir
  233: function _openTrade(TradeParams memory params) internal r

src\LiquidityPool.sol
  379: function orderFee(int256 sizeDelta) public view override
  399: function baseAssetPrice() public view override returns (u
  409: function getSkew() external view override returns (int25(
  430: function openLong(uint256 amount, address user, bytes32 r
  720: function _getSkew() internal view returns (int256 skew) {
  727: function _getDelta() internal view returns (uint256 delta
  798: function _hedge(int256 size, bool isLiquidation) internal

src\ShortCollateral.sol
  121: function liquidate(uint256 positionId, uint256 debt, addr
  153: function getMinCollateral(uint256 shortAmount, address cc
  176: function getLiquidationBonus(address collateral, uint256
  192: function canLiquidate(uint256 positionId) public view ove
  215: function maxLiquidatableDebt(uint256 positionId) public v
```

[16] MISSING NATSPEC COMMENTS

NatSpec comments provide rich code documentation. The following functions miss NatSpec comments. Please consider adding NatSpec comments for these functions.

```
src\LiquidityToken.sol
  19: function refresh() public {
  28: function mint(address _user, uint256 _amt) public onlyPool
  32: function burn(address _user, uint256 _amt) public onlyPool

src\PowerPerp.sol
  22: function refresh() public {
  32: function mint(address _user, uint256 _amt) public onlyExch
  36: function burn(address _user, uint256 _amt) public onlyExch
  40: function transfer(address _to, uint256 _amount) public ove
  45: function transferFrom(address _from, address _to, uint256

src\ShortToken.sol
  33: function refresh() public {
  44: function tokenURI(uint256 tokenId) public view override re
  48: function adjustPosition(
  92: function transferFrom(address _from, address _to, uint256
  97: function safeTransferFrom(address _from, address _to, uint
  102: function safeTransferFrom(address _from, address _to, uin

src\SynthetixAdapter.sol
  18: function getSynth(bytes32 key) public view override return
  22: function getCurrencyKey(address synth) public view overric
  26: function getAssetPrice(bytes32 key) public view override r

src\SystemManager.sol
  62: function init(
  84: function refreshSynthetixAddresses() public {
  89: function setStatusFunction(bytes32 key, bool status) publi

src\VaultToken.sol
  27: function mint(address _user, uint256 _amt) external onlyVa
  31: function burn(address _user, uint256 _amt) external onlyVa
  35: function setVault(address _vault) external {

src\libraries\SignedMath.sol
  5: function signedAbs(int256 x) internal pure returns (int256)
  9: function abs(int256 x) internal pure returns (uint256) {
  13: function max(int256 x, int256 y) internal pure returns (ir
```

```
17: function min(int256 x, int256 y) internal pure returns (ir
```

Dravee (judge) commented:

> **[01]: LACK OF LIMITS FOR SETTING FEES**
> `Low`

> **[02]: SOME FUNCTIONS DO NOT FOLLOW CHECKS-EFFECTS-INTERACTIONS PATTERN**
> `Low` (would’ve been `Refactor` without the detailed explanation and comparison)

> **[03]: nonReentrant MODIFIER CAN BE PLACED AND EXECUTED BEFORE OTHER MODIFIERS IN FUNCTIONS**
> `Non-Critical`

> **[04]: REDUNDANT return KEYWORDS IN ShortToken.transferFrom and ShortToken.safeTransferFrom FUNCTIONS**
> Valid `Refactor` and interesting one at that (good signal)

> **[05]: CONSTANTS CAN BE USED INSTEAD OF MAGIC NUMBERS**
> `Refactor`

> **[06]: HARDCODED STRING THAT IS REPEATEDLY USED CAN BE REPLACED WITH A CONSTANT**
> `Refactor`

> **[07]: ShortToken.adjustPosition FUNCTION DOES NOT NEED TO UPDATE totalShorts and position.shortAmount IN CERTAIN CASE**
> `Refactor` / `Gas` (good one)

> **[08]: LiquidityPool.withdraw FUNCTION CALLS BOTH SUSD.safeTransfer AND SUSD.transfer**
> Would’ve said `Out-of-Scope` but additional details make this relevant.
> `Refactor`.

> **[09]: LiquidityPool.orderFee FUNCTION CAN CALL getMarkPrice() INSTEAD OF exchange.getMarkPrice()**

> `Refactor`, good one.

> Rest is generic `Refactor`.

> Good report with a good signal.

> Also including [Issue 228](#) from the warden as Low.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.