SHERLOCK

Security Review For Flat Money

Private Audit Contest Prepared For:	**Flat Money**
Lead Security Expert:	**xiaoming90**
Date Audited:	**January 6 - January 15, 2025**
Final Commit:	**ac4b406**

Introduction

The Flat Money v2 protocol update allows users to create a market with different collateral and market assets, enabling the design of bespoke derivative products, including perpetuals, options or any imaginable payoff structure. LPs are the counterparty to the traders.

Scope

Repository: dhedge/flatcoin-v1

Audited Commit: 957489baee10d52d65c50ec88f73189b62c36853

Final Commit: ac4b4068a790099910f853d972ec12b641d6b2f8

Files:

- src/FlatcoinVault.sol
- src/LeverageModule.sol
- src/LiquidationModule.sol
- src/OptionsControllerModule.sol
- src/OracleModule.sol
- src/OrderAnnouncementModule.sol
- src/OrderExecutionModule.sol
- src/PerpControllerModule.sol
- src/PositionSplitterModule.sol
- src/StableModule.sol
- src/abstracts/ControllerBase.sol
- src/abstracts/ERC20LockableUpgradeable.sol
- src/abstracts/FeeManager.sol
- src/abstracts/InvariantChecks.sol
- src/abstracts/KeeperFeeBase.sol
- src/abstracts/ModuleUpgradeable.sol
- src/abstracts/OracleModifiers.sol
- src/abstracts/ViewerBase.sol
- src/misc/ArbKeeperFee.sol
- src/misc/ETHCrossAggregator.sol
- src/misc/FlatZapper/FlatZapper.sol

- src/misc/FlatZapper/FlatZapperStorage.sol
- src/misc/OPKeeperFee.sol
- src/misc/OptionViewer.sol
- src/misc/PerpViewer.sol
- src/misc/Swapper/RouterProcessor.sol
- src/misc/Swapper/RouterProcessorStorage.sol
- src/misc/Swapper/Swapper.sol
- src/misc/Swapper/TokenTransferMethods.sol
- src/misc/Swapper/TokenTransferMethodsStorage.sol

Final Commit Hash

ac4b4068a790099910f853d972ec12b641d6b2f8

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
3	18

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

000000	Kirkeelee	vinica_boy
0x37	KupiaSec	xiaoming90
0xc0ffEE	aslanbek	zarkk01
Afriaudit	newspacexyz	
Bigsam	santipu_	

Issue H-1: Incorrect profitLossTotal calculation in option market

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/60

Found by

0x37, KupiaSec, santipu_

Summary

In optionMarket, we use the globalPosition.averagePrice global average price to calculate the `profitLossTotal`. This is incorrect. Because each position's PnL is non-linear.

Root Cause

In OptionsController:53, we calculate the profitLossTotal() based on the current price. We will use this to calculate current LP share's price.

The problem is that it's incorrect to calculate the total PnL using the average price. Option market is a little different with perp market.

Take one example as below:

1. Alice opens one leverage position when the collateral price is 3000 at timestamp T1. The position size is 10.
2. The collateral price increases to 4000 at timestamp T2.
3. Bob opens one leverage position at timestamp T2 with the same position size. Now the global average price becomes 3500.
4. The collateral price drops to 3400 at the timestamp T3. Note: this is one option market. Both positions will be healthy and should not be liquidated.
5. Cathy wants to deposit some collateral as LP. We will calculate current total PnL via `profitLossTotal` in OptionControllerModule. Because the price is less than the average price, the total PnL will be 0.
6. But actually, if we calculate Alice and Bob's position's PnL, we will find out that Alice has some positive PnL and Bob's PnL is 0, and the total PnL should be positive.

```
function profitLossTotal(uint256 price_) public view override returns (int256 pnl_)
↪  {
    FlatcoinVaultStructs.GlobalPositions memory globalPosition =
↪  vault.getGlobalPositions();
```

```
    int256 priceShift = price_ > globalPosition.averagePrice
        ? int256(price_) - int256(globalPosition.averagePrice)
        : int256(0);

    return (int256(globalPosition.sizeOpenedTotal) * (priceShift)) / int256(price_);
}
```

```
function profitLoss(
    LeverageModuleStructs.Position memory position_,
    uint256 price_ // current price.
) public pure override returns (int256 pnl_) {
    int256 priceShift = price_ > position_.averagePrice
        ? int256(price_) - int256(position_.averagePrice)
        : int256(0);
    // price * size = profit. Note: we return the collateral amt as the profit/Loss.
    return (int256(position_.additionalSize) * (priceShift)) / int256(price_);
}
```

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

1. Alice opens one leverage position when the collateral price is 3000 at timestamp T1. The position size is 10.
2. The collateral price increases to 4000 at timestamp T2.
3. Bob opens one leverage position at timestamp T2 with the same position size. Now the global average price becomes 3500.
4. The collateral price drops to 3400 at the timestamp T3. Note: this is one option market. Both positions will be healthy and should not be liquidated.
5. Cathy wants to deposit some collateral as LP. We will calculate current total PnL via `profitLossTotal` in OptionControllerModule. Because the price is less than the average price, the total PnL will be 0.
6. But actually, if we calculate Alice and Bob's position's PnL, we will find out that Alice has some positive PnL and Bob's PnL is 0, and the total PnL should be positive.

Impact

profitLossTotal calculation is incorrect. This will cause the incorrect LP share's price. New LPs will mint shares with one incorrect share prices.

PoC

N/A

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/493

Issue H-2: Announcement Funds Lost Due to a Close Position

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/71

Found by

000000, 0x37, KupiaSec, newspacexyz, santipu_, zarkk01

Summary

When a user has a position closed or liquidated, LeverageModule::burn will delete any pending order the user has without returning the funds.

Root Cause

In LeverageModule.sol::381, the pending announcement order of the user is deleted without returning the funds.

```
  function burn(uint256 tokenId_) public onlyAuthorizedModule {
      IOrderAnnouncementModule orderAnnouncementModule = IOrderAnnouncementModule(
          vault.moduleAddress(FlatcoinModuleKeys._ORDER_ANNOUNCEMENT_MODULE_KEY)
      );

>>    orderAnnouncementModule.deleteOrder(ownerOf(tokenId_));
      orderAnnouncementModule.deleteLimitOrder(tokenId_);

      _burn(tokenId_);
  }
```

Internal pre-conditions

- A user has a position closed or liquidated while having a pending announced order

External pre-conditions

No response

Attack Path

1. A user opens a leverage position.

2. After some days, the price of rETH decreases.
3. The user notices that his position is going to be liquidated, but he doesn't want that to happen. So, he announces an `announcementAdjust` order to add more margin and avoid liquidation.
4. Before the `announcementAdjust` is executed, the position gets liquidated, and therefore the position is closed.
5. When the `tokenID` linked to the liquidated position is burned, all the announcement orders linked to the account of that `tokenID` are automatically deleted.
6. Since the `announcementAdjust` is deleted, it cannot be executed nor canceled. The margin linked to the `announcementAdjust` is locked in the execution module, and there is no way to recover it, effectively losing the margin funds.

A variant of this attack can be the following:

1. Bob announces an order to open a long position
2. Alice sees that and announces an order to open another long position on behalf of Bob with the minimum funds possible, along with a limit order that will be executed immediately.
3. Alice executes her order first so the minimal long position is created for Bob, along with a limit order that will be executed.
4. Alice waits a few seconds more and executes the limit position linked to Bob's minimal long position, causing the removal of Bob's original announced order, causing a loss of funds for Bob.
    - This attack relies that Alice's order can be executed before Bob's, which may not always be the case but it's a possibility.

Another variant of the previous attack would be to use the reentrancy on `safeMint` to transfer the recently created minimal position to Bob, and later execute the limit order to delete any order that Bob has pending, causing a loss of funds for him.

Impact

The user loses all the funds related to the pending announced order.

Here are the detailed loss of funds for each announced order a user could have pending:

- **Stable Deposit**: The funds lost will be the deposit amount and the keeper fee.
- **Stable Withdraw:** When announcing the order, the tokens to be withdrawn will be locked and will be kept locked forever when the issue is triggered.
- **Leverage Open:** The funds lost will be the margin and the fees (trade fee and keeper fee).

- **Leverage Adjust:** When the user wants to add funds to the margin, those funds for the extra margin will be locked, along with the keeper fee.

PoC

You can paste the follwing PoC to `test/unit/Liquidation-Module/Liquidate.t.sol`:

```
function test_announceOrder_deleted_by_liquidation() public {
    vm.startPrank(alice);
    setCollateralPrice(1000e8);
    announceAndExecuteDeposit({
        traderAccount: alice,
        keeperAccount: keeper,
        depositAmount: 100e18,
        oraclePrice: 1000e8,
        keeperFeeAmount: 0
    });

    // Alice opens a position with 50 collateralAsset margin
    uint256 tokenId = announceAndExecuteLeverageOpen({
        traderAccount: alice,
        keeperAccount: keeper,
        margin: 50e18,
        additionalSize: 100e18,
        oraclePrice: 1000e8,
        keeperFeeAmount: 0
    });

    skip(2 days);

    // Prices decreases to liquidation
    uint256 liqPrice = viewer.liquidationPrice(0);
    uint256 newCollateralPrice = (liqPrice - 1e18) / 1e10;
    setCollateralPrice(newCollateralPrice);
    assertTrue(liquidationModProxy.canLiquidate(tokenId));

    // Alice wants to adjust her position to not get liquidated
    announceAdjustLeverage({
        traderAccount: alice,
        tokenId: tokenId,
        marginAdjustment: 50e18,
        additionalSizeAdjustment: 0,
        keeperFeeAmount: 0
    });

    // Liquidate the position before the announcement order gets executed
    vm.startPrank(liquidator);
    bytes[] memory priceUpdateData = getPriceUpdateData(newCollateralPrice);
    liquidationModProxy.liquidate{value: 2}(tokenId, priceUpdateData);
    LeverageModuleStructs.Position memory position =
    ↪ vaultProxy.getPosition(tokenId);

    // The position should be liquidated
    assertEq(position.marginDeposited, 0);

    // The execution should revert as the announcement doesn't exist
    vm.expectRevert();
    vm.startPrank(keeper);
    orderExecutionModProxy.executeOrder{value: 1}(alice, priceUpdateData);

    // Pass time till the order has expired to try to cancel it
    skip(uint256(orderAnnouncementModProxy.minExecutabilityAge()));

    // Extract keeper fee
    uint256 keeperFee = mockKeeperFee.getKeeperFee();

    uint256 aliceBalanceBefore = collateralAsset.balanceOf(alice);
    // Canceling the announced order should do nothing because the order
    ↪ announcement doesn't exist, should also not revert
    vm.startPrank(alice);
    orderExecutionModProxy.cancelExistingOrder(alice);
    uint256 aliceBalanceAfter = collateralAsset.balanceOf(alice);
    assertEq(aliceBalanceBefore, aliceBalanceAfter);
    // The 50e18 + keeper fee of Alice should be locked in the execution module
    ↪ with no announcement order to execute
    assertEq(collateralAsset.balanceOf(address(orderExecutionModProxy)),
    ↪ 50e18+keeperFee);
}
```

Mitigation

To mitigate this issue is recommended to not directly delete a pending order but to cancel it, along with returning the funds of the pending announced order.

To achieve this, an extra function could be created to cancel an order without checking for `maximumExecutabilityTime`, allowing it to be canceled even if it hasn't expired. This new function should only be called by authorized modules

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/497

Issue H-3: The `StableModule.executeWithdraw()` function will always revert due to an integer underflow during the check for significant impact on the stable token price when the collateral has a low decimal value and a high price, such as WBTC.

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/94

Found by

000000, KupiaSec, santipu_

Summary

The `StableModule.executeWithdraw()` function checks if there is no significant impact on stable token price. However, when the collateral has a low decimal value and a high price, such as WBTC, it will always revert due to an integer underflow during the check.

Root Cause

The `StableModule.executeWithdraw()` function checks that there is no change greater than 1e6 on the stable token price. However, if the `stableCollateralPerShareBefore` is less than 1e6, this check will always revert due to an integer underflow.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/StableModule.sol#L130-L133

```
            if (
131:            stableCollateralPerShareAfter < stableCollateralPerShareBefore -
 ↪  1e6 ||
                stableCollateralPerShareAfter > stableCollateralPerShareBefore + 1e6
            ) revert PriceImpactDuringWithdraw();
```

For example, the current price of 1 WBTC is about 1e5USD, so the initial `stableCollateralPerShare` of WBTC is about `(1e18* 1e8)/(1e18*1e5)=1e3`. (The decimal of WBTC is 8, and the decimal of collateral price is 18 in the `OracleModule.sol`.)

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/OracleModule.sol#L189

```
@>      price_ = uint256(price) * (10 ** 10); // convert Chainlink oracle decimals
 ↪  8 -> 18
```

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/StableModule.sol#L235

```
@>      collateralPerShare_ = (1e18 * (10 ** collateral.decimals())) /
 ↪  collateralPrice;
```

While the `collateralPerShare` will change, the likelihood of it exceeding `1e6` is very low. Therefore, the `StableModule.executeWithdraw()` function will always revert at L131. As a result, collaterals with a low decimal value and a high price, such as WBTC, cannot be withdrawn, resulting in the funds being stuck.

Internal pre-conditions

none

External pre-conditions

none

Attack Path

none

Impact

Users cannot withdraw their funds.

PoC

none

Mitigation

The check that there is no significant impact on stable token price should be improved.

```
             if (
-                stableCollateralPerShareAfter < stableCollateralPerShareBefore -
 ↪  1e6 ||
+                stableCollateralPerShareAfter + 1e6 <
 ↪  stableCollateralPerShareBefore ||
                 stableCollateralPerShareAfter > stableCollateralPerShareBefore + 1e6
             ) revert PriceImpactDuringWithdraw();
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/519

Issue M-1: Arguments order mismatch in PositionSplitterModule.split() for leverage check

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/3

Found by

KupiaSec, newspacexyz

Summary

In the PositionSplitterModule.split() function, the arguments passed to leverageModule.checkLeverageCriteria() are in the wrong order. This leads to incorrect leverage calculations, which can result in invalid positions passing leverage checks or valid positions failing them. The issue affects the core functionality of position splitting and can lead to unexpected behavior, including protocol insolvency or bad debt creation.

Root Cause

The function checkLeverageCriteria() in the LeverageModule expects the parameters in the order: https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/LeverageModule.sol#L429 However, in PositionSplitterModule.split(), the arguments are passed in reverse order: https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/PositionSplitterModule.sol#L114 https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/PositionSplitterModule.sol#L137 This causes the leverage calculation to be based on incorrect input values.

Internal pre-conditions

The LeverageModule requires margin_ as the first argument and size_ as the second argument for checkLeverageCriteria().

External pre-conditions

The split() function in PositionSplitterModule is called with a valid position and a fraction for splitting the position.

Attack Path

1. A user calls the split() function to split a position.
2. Due to the reversed arguments in checkLeverageCriteria(), the leverage calculation is incorrect.
3. This can:

- Allow splitting into invalid positions that do not meet leverage criteria, or
- Incorrectly block splitting valid positions by failing leverage checks.

4. If invalid positions are created, they may immediately become liquidatable or lead to incorrect margin and size balances, destabilizing the protocol.

Impact

Invalid positions passing leverage checks can destabilize the system by creating positions that are improperly collateralized or liquidatable, leading to bad debt. Improperly collateralized positions can lead to insolvency if they cannot be liquidated effectively.

PoC

1. Set up a position with `marginDeposited = 100` and `additionalSize = 200`.
2. Call the `split()` function with `positionFraction_ = 0.5e18` (50%).
3. Observe that the leverage check fails or passes incorrectly due to the argument order mismatch.

```
// Assume leverageMin = 1.5e18 and leverageMax = 5e18
leverageModule.checkLeverageCriteria(200, 100); // Incorrect order
```

This results in:

- Leverage = (200 + 100) * 1e18 / 200 = 1.5, which is invalid.
- Correct calculation should be: (100 + 200) * 1e18 / 100 = 3.0, which is valid.

Mitigation

Fix the Argument Order: Update the argument order in PositionSplitterModule.split() to:

```
leverageModule.checkLeverageCriteria(primaryMargin, primarySize);
leverageModule.checkLeverageCriteria(newPositionMargin, newPositionSize);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/498

Issue M-2: No min/max price check is problematic

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/6

The protocol has acknowledged this issue.

Found by

000000, Kirkeelee, aslanbek, zarkk01

Summary

No min/max price check

Root Cause

When calling ETHCrossAggregator.latestRoundData(), we fetch the ETH/USD price. However, we do not check against the min/max price of the feed. Despite it being deprecated, some feeds still have min and max prices set, such is the ETH/USD feed (https://arbiscan.io/address/0x3607e46698d218B3a5Cae44bF381475C0a5e2ca7#readContract). Not checking against it can be problematic and could result in wrong accounting.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Incorrect price which causes wrong accounting

PoC

No response

Mitigation

Fetch and check the min and max price

Issue M-3: Some Users will not be able to Adjust their Position because of the wrong settlement valuation when Announcing Positions adjustments

Source: https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/20

The protocol has acknowledged this issue.

Found by

Bigsam

Summary

Margin adjustment is a time-sensitive function and users who are unable to do this risk being liquidated. A wrong assessment of the users Margin will cause the entire system to revert when it should not, the PNL at the present price will be lower than the pnl at the maxfill.

Root Cause

No response

Internal pre-conditions

```


        if (additionalSizeAdjustment_ >= 0) {
                    // If additionalSizeAdjustment equals zero, trade fee is zero as
    ↪   well
                    // and no need to check for skew max.
                    if (additionalSizeAdjustment_ > 0) {
                        tradeFee =
    ↪   FeeManager(address(vault)).getTradeFee(uint256(additionalSizeAdjustment_));
                        vault.checkSkewMax({
                            sizeChange: uint256(additionalSizeAdjustment_),
                            stableCollateralChange: int256(tradeFee -
    ↪   FeeManager(address(vault)).getProtocolFee(tradeFee))
                        });
                    }

@audit>>                    if (fillPrice_ < currentPrice) revert
    ↪   MaxFillPriceTooLow(fillPrice_, currentPrice);
        } else {
                    tradeFee =
    ↪   FeeManager(address(vault)).getTradeFee(uint256(additionalSizeAdjustment_ * -1));

@audit>>               if (fillPrice_ > currentPrice) revert
    ↪   MinFillPriceTooHigh(fillPrice_, currentPrice);

          }

{
           // New additional size will be either bigger or smaller than current
    ↪   additional size
           // depends on if additionalSizeAdjustment is positive or negative.
           int256 newAdditionalSize =
    ↪   int256(vault.getPosition(tokenId_).additionalSize) + additionalSizeAdjustment_;

           // If user withdraws margin or changes additional size with no changes
    ↪   to margin, fees are charged from their existing margin.

    @audit>> summary >>            int256 newMarginAfterSettlement =
    ↪   leverageModule.getPositionSummary(tokenId_).marginAfterSettlement +
                    ((marginAdjustment_ > 0) ? marginAdjustment_ : marginAdjustment_ -
    ↪   int256(totalFee));


      // New margin or size can't be negative, which means that they want to withdraw
    ↪   more than they deposited or not enough to pay the fees

    @audit>> points of possible reversal >>       if (newMarginAfterSettlement < 0 ||
    ↪   newAdditionalSize < 0)
                    revert
    ↪   ICommonErrors.ValueNotPositive("newMarginAfterSettlement|newAdditionalSize");


    @audit>> points of possible reversal >>             if (
                    ILiquidationModule(vault.moduleAddress(FlatcoinModuleKeys._LIQUIDAT⌋
    ↪   ION_MODULE_KEY))
                        .getLiquidationMargin(uint256(newAdditionalSize), fillPrice_)
    ↪   >= uint256(newMarginAfterSettlement)
           ) revert ICommonErrors.PositionCreatesBadDebt();


  // New values can't be less than min margin and min/max leverage requirements.

    @audit>> points of possible reversal >>
    ↪   leverageModule.checkLeverageCriteria(uint256(newMarginAfterSettlement),
    ↪   uint256(newAdditionalSize));
            }
```

The Current MarginSize return is against the current price instead of against the fill Price

```
@audit >>.  function getPositionSummary(
        uint256 tokenId_
    ) external view returns (LeverageModuleStructs.PositionSummary memory
↪   positionSummary_) {

@audit >>.        (uint256 currentPrice, ) = IOracleModule(vault.moduleAddress(Flat⌋
↪   coinModuleKeys._ORACLE_MODULE_KEY)).getPrice(
            address(vault.collateral())
        );

@audit >>. current price is used not fill>>         return
↪   getPositionSummary(vault.getPosition(tokenId_), currentPrice);

  }
```

From the code it can be seen that the liquidation fee is calculated USING the fillprice for accuracy but this use the wrong check thereby causing an unnecessary revert.

```
@audit >>.     if (
                ILiquidationModule(vault.moduleAddress(FlatcoinModuleKeys._LIQUIDAT⌋
↪   ION_MODULE_KEY))
                    .getLiquidationMargin(uint256(newAdditionalSize), fillPrice_)
↪   >= uint256(newMarginAfterSettlement)
            ) revert ICommonErrors.PositionCreatesBadDebt();
```

External pre-conditions

No response

Attack Path

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/OrderAnnouncementModule.sol#L389-L391

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/OrderAnnouncementModule.sol#L397-L400

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/LeverageModule.sol#L391-L398

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/LiquidationModule.sol#L240

Impact

Legitimate calls from mostly postions lossing funds to increase margin or reduce leverage will revert either at the liquidation check of minleverage/maxleverage/minmargin check causing a DOS to a time sensitive function.

PoC

```
function test_adjust_position_margin_increase_will_revert_an_healthy_position()
↪  public {
     uint256 aliceCollateralBalanceBefore = collateralAsset.balanceOf(alice);
     uint256 stableDeposit = 100e18;
     uint256 collateralPrice = 1000e8;
     uint256 keeperFee = mockKeeperFee.getKeeperFee();

     announceAndExecuteDeposit({
         traderAccount: alice,
         keeperAccount: keeper,
         depositAmount: stableDeposit,
         oraclePrice: collateralPrice,
         keeperFeeAmount: keeperFee
     });

     // 10 ETH margin, 30 ETH size (4x)
     uint256 tokenId = announceAndExecuteLeverageOpen({
         traderAccount: alice,
         keeperAccount: keeper,
         margin: 10e18,
         additionalSize: 30e18,
         oraclePrice: collateralPrice,
         keeperFeeAmount: keeperFee
     });





   uint256 liqPrice = viewer.liquidationPrice(0);



 LeverageModuleStructs.Position memory position = vaultProxy.getPosition(tokenId);
 LeverageModuleStructs.PositionSummary memory positionSummary
↪  =leverageModProxy.getPositionSummary(position, (liqPrice));

 emit log_named_int("Profitloss", positionSummary.profitLoss);
 emit log_named_int("accrue", positionSummary.accruedFunding);
 emit log_named_int("Margin at liquidation", positionSummary.marginAfterSettlement);
 emit log_named_uint("Liquidation price", liqPrice);


uint premargin =  liquidationModProxy.getLiquidationMargin(31e18, liqPrice);
 emit log_named_uint("liquidation margin at liquidation price", premargin);



 LeverageModuleStructs.Position memory position2 = vaultProxy.getPosition(tokenId);
 LeverageModuleStructs.PositionSummary memory positionSummary2
↪   =leverageModProxy.getPositionSummary(position2, ((liqPrice + 1e18) - ((liqPrice
↪   + 1e18)/1000)));

 emit log_named_int("Profitloss2", positionSummary2.profitLoss);
 emit log_named_int("accrue2", positionSummary2.accruedFunding);
 emit log_named_int("Current price Margin", positionSummary2.marginAfterSettlement);
 emit log_named_uint("current Price Now", ((liqPrice + 1e18) - ((liqPrice +
↪   1e18)/1000)));



   uint256 newCollateralPrice = ((liqPrice + 1e18) - ((liqPrice + 1e18)/1000));


      setCollateralPrice(newCollateralPrice/1e10);







     uint256 keeperFee2 = mockKeeperFee.getKeeperFee();


   vm.startPrank(alice);

     collateralAsset.approve(address(orderAnnouncementModProxy), 6e18);





     emit log_named_uint("fill price", liqPrice+1e18);
```

```
      uint premargin5 =  liquidationModProxy.getLiquidationMargin(31e18,
↪  (liqPrice+1e18));
      emit log_named_uint("Liquidation Margin calculated during annoucement using
↪  fillprice", premargin5);




      LeverageModuleStructs.Position memory position5 =
↪  vaultProxy.getPosition(tokenId);
     LeverageModuleStructs.PositionSummary memory positionSummary5
↪  =leverageModProxy.getPositionSummary(position5,liqPrice+1e18);

       emit log_named_int("Profitloss5", positionSummary5.profitLoss);
      emit log_named_int("accrue5", positionSummary5.accruedFunding);
       emit log_named_int("Actual Margin at Fill Price",
↪  positionSummary5.marginAfterSettlement);

       assert(premargin5 > uint(positionSummary2.marginAfterSettlement));

        assert(premargin5 > uint(positionSummary2.marginAfterSettlement));
        assert(uint(positionSummary5.marginAfterSettlement) > premargin5);

 }
```

```
Logs:
Profitloss: -975000000000000000
accrue: 0
Margin at liquidation: 25000000000000000
Liquidation price: 754716981132075471698
liquidation margin at liquidation price: 28750000000000000
Profitloss2: -9737138081831363233
accrue2: 0
Current price Margin: 262861918168636767
current Price Now: 754961264150943396227
fill price: 755716981132075471698
liquidation Margin calculated during annoucement using fillprice: 287324669812498439
Profitloss5: -9697400943749531870
accrue5: 0
Actual Margin at Fill Price: 302599056250468130


Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.99ms (2.86ms CPU
↪  time)
```

Based on the last assertions in the test the Margin at fill is higher than the Liquidation margin and trade should be announced but the use of the current price will revert the transaction, look at assertion 2.

Mitigation

Use the Fillprice to calculate the Margin at the point of settlement

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/LeverageModule.sol#L405-L423

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/503

Issue M-4: Attacker will cause a permanent DoS to all new leverage positions being created

Source: https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/24

Found by

santipu_

Summary

Anyone can cause a permanent DoS on all leverage orders by announcing a stable deposit on behalf of `address(0)`.

Root Cause

In LeverageModule.sol::462 when a leverage position NFT is being minted, it checks if the address zero has any pending orders, reverting if that's the case.

```
    function _update(address to_, uint256 tokenId_, address auth_) internal virtual
 ↪  override returns (address from) {
        // ...

>>      DelayedOrderStructs.Order memory normalOrder =
 ↪  orderAnnouncementModule.getAnnouncedOrder(_ownerOf(tokenId_));
        DelayedOrderStructs.Order memory limitOrder =
 ↪  orderAnnouncementModule.getLimitOrder(tokenId_);

        // If a normal order (leverage adjust/close) exists, disallow the transfer.
>>      if (normalOrder.orderType != DelayedOrderStructs.OrderType.None) {
            revert ICommonErrors.OrderExists(normalOrder.orderType);
        }

        // ...
    }
```

When a `tokenId` is about to be minted, the function `_ownerOf` will return `address(0)`, therefore it will get any pending orders the address zero has, reverting the transaction if there is a pending order on behalf of `address(0)`.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Attacker announces a stable deposit on behalf of address zero.
2. When a normal user wants to execute an order to create a leverage position, it will always revert in `LeverageModule::_update` due to checking if the address zero has any pending orders.

Impact

Anyone can cause a permanent DoS on the creation of new leverage positions by just announning a stable deposit on behalf of address zero. Given that no more leverage positions can be created, the protocol cannot function correctly and it would have to be redeployed again, migrating all of users' funds.

Overall, this issue breaks core contract functionality, rendering the contract useless, hence the medium severity.

The DoS will be permanent because the order on behalf of address zero cannot be canceled. That is because `cancelExistingOrder` would revert when trying to transfer the tokens back to `address(0)`, as most ERC20 implementations revert when trying to transfer tokens to `address(0)`

PoC

The following PoC can be pasted and run in `Stable-Module/Deposit.t.sol`. The test demonstrates the attack, and is **expected to revert** on the last call with the error `OrderExists(1)`, which shouldn't be the case as Alice does not have any pending orders.

```
function test_address_zero_blocks_orders() public {
    // We open an order for address(0) to set the trap
    announceStableDepositFor({
        traderAccount: keeper,
        receiver: address(0),
        depositAmount: 1e18,
        keeperFeeAmount: 0
    });

    // A normal user deposits
    announceAndExecuteDeposit({
        traderAccount: bob,
        keeperAccount: keeper,
        depositAmount: 100e18,
        oraclePrice: 1000e8,
        keeperFeeAmount: 0
    });

    // We now try to open other leverage positions
    // It reverts here with `OrderExists(1)`
    announceAndExecuteLeverageOpen({
        traderAccount: alice,
        keeperAccount: keeper,
        margin: 60e18,
        additionalSize: 100e18,
        oraclePrice: 1000e8,
        keeperFeeAmount: 0
    });
}
```

Mitigation

To mitigate this issue, is recommended to avoid the checks in `LeverageModule::_update` when the NFT is being minted:

```
    function _update(address to_, uint256 tokenId_, address auth_) internal virtual
 ↪  override returns (address from) {
        IOrderAnnouncementModule orderAnnouncementModule = IOrderAnnouncementModule(
            vault.moduleAddress(FlatcoinModuleKeys._ORDER_ANNOUNCEMENT_MODULE_KEY)
        );

+       if (_ownerOf(tokenId_) != address(0)) {

        DelayedOrderStructs.Order memory normalOrder =
 ↪  orderAnnouncementModule.getAnnouncedOrder(_ownerOf(tokenId_));
        DelayedOrderStructs.Order memory limitOrder =
 ↪  orderAnnouncementModule.getLimitOrder(tokenId_);

        // If a normal order (leverage adjust/close) exists, disallow the transfer.
        if (normalOrder.orderType != DelayedOrderStructs.OrderType.None) {
            revert ICommonErrors.OrderExists(normalOrder.orderType);
        }

        // If a limit order exists, disallow the transfer.
        if (limitOrder.orderType != DelayedOrderStructs.OrderType.None) {
            revert ICommonErrors.OrderExists(limitOrder.orderType);
        }

+       }

        return super._update(to_, tokenId_, auth_);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/500

Issue M-5: Attackers can steal residue/donated tokens in the Swapper contract

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/27

Found by

Bigsam

Summary

According to the approval action design in the Router Processor, it is believed that the swapper should not hold any tokens after a swap but this assumption can fail in 2 conditions.
SwapData passed to the router are never verified and there are two different ways of swapping, EXACT-IN and EXACT-OUT. Using exact Out will cause some percentage of the amount in ARE NOT used from the contract during the swap, this funds are not sent back to the msg.sender. Although there is a rescue function to this effect to rescue DONATED / RESIDUAL TOKENS this doesn't take away a possible Race condition that could see an attacker successfully pass in data in the swapper to use this tokens and receive free tokens even open positions.

Root Cause

Residual/Donated tokens in the SWAPPER.sol contract

1. Aggregator.SwapDatas are not verified, Amount in encoded can be greater than the amount in passed.
2. No check after each swap to ensure the token left if any are sent back to the msg.sender.
3. No Pause/emergency mechanism in the Swapper contract.
4. Max approval is given to the router so nothing stops this attack.

Internal pre-conditions

```
function profitLossTotal(uint256 price_) public view override returns (int256 pnl_)
↪  {
    FlatcoinVaultStructs.GlobalPositions memory globalPosition =
↪   vault.getGlobalPositions();

    int256 priceShift = price_ > globalPosition.averagePrice
```

```
        ? int256(price_) - int256(globalPosition.averagePrice)
        : int256(0);

    return (int256(globalPosition.sizeOpenedTotal) * (priceShift)) / int256(price_);
}
```

```
function profitLoss(
    LeverageModuleStructs.Position memory position_,
    uint256 price_ // current price.
) public pure override returns (int256 pnl_) {
    int256 priceShift = price_ > position_.averagePrice
        ? int256(price_) - int256(position_.averagePrice)
        : int256(0);
    // price * size = profit. Note: we return the collateral amt as the profit/Loss.
    return (int256(position_.additionalSize) * (priceShift)) / int256(price_);
}
```

No check to ensure that indeed there are no tokens/ funds in the contract . No decoding of swap data to verify amount-in

```
/// @notice Swap tokens using the given `swapStruct_`.
   /// @dev Only supports SINGLE_IN_SINGLE_OUT and MULTI_IN_SINGLE_OUT swap types.
   /// @param swapStruct_ The struct containing all the data required to process
↪  the swap(s).
   function swap(SwapperStructs.InOutData calldata swapStruct_) external payable {
       uint256 destAmountBefore =
↪  swapStruct_.destData.destToken.balanceOf(address(this));

       // Transfer all the `srcTokens` to this contract.
       _transferFromCaller(msg.sender, swapStruct_.srcData);

       // Process swaps based on `srcData` array.
       // The first loop iterates over the `srcData` array. The number of
↪  iterations is equal to the number of transfer methods used in the swap.
       // For example if the swap uses `TokenTransferMethod.ALLOWANCE` for all
↪  `srcTokens`, then the outer loop will iterate only once.
       // If the swap uses `TransferMethod.ALLOWANCE` for the first x `srcTokens`
↪  and `TransferMethod.PERMIT` for the next y `srcTokens`,
       // then the outer loop will iterate twice.
       for (uint256 i; i < swapStruct_.srcData.length; ++i) {
           // The second loop iterates over the `srcTokens` array in which the
↪  `srcTokens` are transferred and swapped using the same token transfer method.
           for (uint256 j; j < swapStruct_.srcData[i].srcTokenSwapDetails.length;
↪  ++j) {
               _processSwap({srcTokenSwapDetails_:
↪  swapStruct_.srcData[i].srcTokenSwapDetails[j]});
           }
       }
```

```
        // Check that we got enough of each `destToken` after processing and
    ↪   transfer them to the caller.
        // Note that we don't consider the current `destToken` balance of this
    ↪   contract as the received amount
        // as the amount can be more than the actual received amount due to someone
    ↪   else transferring tokens to this contract.
        // The following approach gives us the ability to rescue funds from this
    ↪   contract.
        uint256 destAmountReceived =
    ↪   swapStruct_.destData.destToken.balanceOf(address(this)) - destAmountBefore;

        if (destAmountReceived < swapStruct_.destData.minDestAmount)
            revert InsufficientAmountReceived(
                swapStruct_.destData.destToken,
                destAmountReceived,
                swapStruct_.destData.minDestAmount
            );

        swapStruct_.destData.destToken.safeTransfer(msg.sender, destAmountReceived);
    }
```

External pre-conditions

Using 1inch as an example

From the 1inchrouter base mainnet ==> https://base.blockscout.com/address/0x1111111254 21cA6dc452d289314280a0f8842A65?tab=contract

```
    struct SwapDescription {
        IERC20 srcToken;
        IERC20 dstToken;
        address payable srcReceiver;
        address payable dstReceiver;
        uint256 amount;
        uint256 minReturnAmount;
        uint256 flags;
    }

    /**
    * @notice Performs a swap, delegating all calls encoded in `data` to
↪   `executor`. See tests for usage examples.
    * @dev Router keeps 1 wei of every token on the contract balance for gas
↪   optimisations reasons.
    *      This affects first swap of every token by leaving 1 wei on the contract.
    * @param executor Aggregation executor that executes calls described in `data`.
    * @param desc Swap description.
    * @param data Encoded calls that `caller` should execute in between of swaps.
    * @return returnAmount Resulting token amount.
    * @return spentAmount Source token amount.
    */
    function swap(
        IAggregationExecutor executor,
        SwapDescription calldata desc,
        bytes calldata data
    )
        external
        payable
        whenNotPaused()
        returns (
            uint256 returnAmount,
            uint256 spentAmount
        )
    {
        if (desc.minReturnAmount == 0) revert ZeroMinReturn();

        IERC20 srcToken = desc.srcToken;
        IERC20 dstToken = desc.dstToken;

        bool srcETH = srcToken.isETH();
        if (desc.flags & _REQUIRES_EXTRA_ETH != 0) {
            if (msg.value <= (srcETH ? desc.amount : 0)) revert
↪   RouterErrors.InvalidMsgValue();
        } else {
            if (msg.value != (srcETH ? desc.amount : 0)) revert
↪   RouterErrors.InvalidMsgValue();
        }

        if (!srcETH) {
            srcToken.safeTransferFromUniversal(msg.sender, desc.srcReceiver,
↪   desc.amount, desc.flags & _USE_PERMIT2 != 0);
        }

        returnAmount = _execute(executor, msg.sender, desc.amount, data);
        spentAmount = desc.amount;

        if (desc.flags & _PARTIAL_FILL != 0) {
            uint256 unspentAmount = srcToken.uniBalanceOf(address(this));
            if (unspentAmount > 1) {
                // we leave 1 wei on the router for gas optimisations reasons
                unchecked { unspentAmount--; }
                spentAmount -= unspentAmount;
                srcToken.uniTransfer(payable(msg.sender), unspentAmount);
            }
            if (returnAmount * desc.amount < desc.minReturnAmount * spentAmount)
↪   revert RouterErrors.ReturnAmountIsNotEnough(returnAmount, desc.minReturnAmount
↪   * spentAmount / desc.amount);
        } else {
            if (returnAmount < desc.minReturnAmount) revert
 ↪  RouterErrors.ReturnAmountIsNotEnough(returnAmount, desc.minReturnAmount);
        }

        address payable dstReceiver = (desc.dstReceiver == address(0)) ?
 ↪  payable(msg.sender) : desc.dstReceiver;
        dstToken.uniTransfer(dstReceiver, returnAmount);
    }

    function _execute(
        IAggregationExecutor executor,
        address srcTokenOwner,
        uint256 inputAmount,
        bytes calldata data
    ) private returns(uint256 result) {
        bytes4 executeSelector = executor.execute.selector;
        assembly ("memory-safe") {  // solhint-disable-line no-inline-assembly
            let ptr := mload(0x40)

   @audit>> selector>>         mstore(ptr, executeSelector)
@audit>>            mstore(add(ptr, 0x04), srcTokenOwner)
  @audit>>          calldatacopy(add(ptr, 0x24), data.offset, data.length)
  @audit>>          mstore(add(add(ptr, 0x24), data.length), inputAmount)

 @audit>>           if iszero(call(gas(), executor, callvalue(), ptr, add(0x44,
 ↪  data.length), 0, 0x20)) {
                returndatacopy(ptr, 0, returndatasize())
                revert(ptr, returndatasize())
            }

            result := mload(0)
        }
    }
}
```

Router -https://github.com/1inch/universal-router/blob/main/contracts/UniversalRouter.sol

```
/// @inheritdoc IUniversalRouter
  function execute(bytes calldata commands, bytes[] calldata inputs) public payable
 ↪  isNotLocked {
      bool success;
      bytes memory output;
      uint256 numCommands = commands.length;
      if (inputs.length != numCommands) revert LengthMismatch();

      // loop through all given commands, execute them and pass along outputs as
 ↪  defined
```

```
        for (uint256 commandIndex = 0; commandIndex < numCommands;) {
            bytes1 command = commands[commandIndex];

            bytes memory input = inputs[commandIndex];

            (success, output) = dispatch(command, input);

            if (!success && successRequired(command)) {
                revert ExecutionFailed({commandIndex: commandIndex, message: output});
            }

            unchecked {
                commandIndex++;
            }
        }
```

List of commands - https://github.com/1inch/universal-router/blob/main/contracts/libraries/Commands.sol

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.17;

/// @title Commands
/// @notice Command Flags used to decode commands
library Commands {
    // Masks to extract certain bits of commands
    bytes1 internal constant FLAG_ALLOW_REVERT = 0x80;
    bytes1 internal constant COMMAND_TYPE_MASK = 0x3f;

    // Command Types. Maximum supported command at this moment is 0x1F.

    // Command Types where value<0x08, executed in the first nested-if block
    uint256 constant V3_SWAP_EXACT_IN = 0x00;
    uint256 constant V3_SWAP_EXACT_OUT = 0x01;
    uint256 constant PERMIT2_TRANSFER_FROM = 0x02;
    uint256 constant PERMIT2_PERMIT_BATCH = 0x03;
    uint256 constant SWEEP = 0x04;
    uint256 constant TRANSFER = 0x05;
    uint256 constant PAY_PORTION = 0x06;
    // COMMAND_PLACEHOLDER = 0x07;

    // Command Types where 0x08<=value<=0x0f, executed in the second nested-if block
    uint256 constant V2_SWAP_EXACT_IN = 0x08;
    uint256 constant V2_SWAP_EXACT_OUT = 0x09;
```

Attack Path

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/misc/Swapper/RouterProcessor.sol#L60-L61

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/misc/Swapper/RouterProcessor.sol#L31-L67

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/misc/Swapper/Swapper.sol#L109-L115

Impact

Residual/Donated tokens can be left in the contract and this tokens can be stolen by an attacker.

PoC

Due to a bug in the API data obtained

I reduced the source token amount and manipulated the swap data, since swap data is not verified and passed directly to the router .

1. reduce Alice source amount in data

```
/// @dev Get the default amount in `token` denomination based on the token's
↪ decimals and globally set `DEFAULT_AMOUNT`.
 function getDefaultAmountInToken(Token memory token) internal view returns
↪  (uint256) {
      return
       (   ( (DEFAULT_AMOUNT * (10 ** (8 +
↪ IERC20Metadata(address(token.token)).decimals()))) /
           _getChainlinkPrice(token.priceFeed)) - 100000);
 }
```

2. Manipulate swap data and hardecode inflated sourceamount.

```
(swapStructArrays.routerKeys[i], minDestAmount, swapStructArrays.swapDatas[i]) =
↪  _getDataFromAggregator(
            alice,
            srcTokens[i].token,
            destToken.token,
           249994250,
            aggregators[I]
        );
```

3. silence the check for swapper balance equals zero in test because we donated to show that residue tokens can be stolen. In testbuilder function

```
// assertTrue(
    //     swapStructArrays.srcTokens[i].token.balanceOf(address(swapperProxy)) ==
↪   0,
    //     "Swapper's src balance should be 0"
    // );
}
```

4. Silence all other tests or create a new area for the above
5. Input RPC and Run - FOUNDRY_PROFILE=integration forge t --match-path test/integration/Base/SwapperAndZapperAllTests.t.sol

```
function test_integration_zapper_rescue_funds() public {
    // Ensure no active prank before starting a new one
    vm.stopPrank();

    // Initialize the source token array with USDC
    Token[] memory srcTokens = TokenArrayBuilder.fill(1, USDC);
    Token memory destToken = rETH;

    // Start impersonating 'bob'
    vm.startPrank(bob);

    // Save Bob's USDC balance before the transfer
    uint256 bobBalanceBefore = USDC.token.balanceOf(bob);

    // Bob transfers USDC directly to the Swapper Proxy
    uint256 transferAmount = 100e6; // Amount in USDC (assuming 6 decimals) //
↪   donation to to if residue tokens can be stolen
    require(USDC.token.balanceOf(bob) >= transferAmount, "Bob has insufficient USDC
↪   balance");
    USDC.token.transfer(address(swapperProxy), transferAmount);

    // Validate that the transfer was successful
    uint256 proxyBalance = USDC.token.balanceOf(address(swapperProxy));
    require(proxyBalance >= transferAmount, "USDC transfer to Swapper Proxy
↪   failed");

     // Stop impersonating 'bob'
    vm.stopPrank();

    // Run all aggregator tests with the specified parameters
    runAllAggregatorTests({
        srcTokens: srcTokens,
        destToken: destToken,
        transferMethod: SwapperStructs.TransferMethod.ALLOWANCE
    });

    emit log_named_uint("How much do we have left in the
↪   contract",USDC.token.balanceOf(address(swapperProxy)));
     emit log_named_uint("Amount stolen by attacker for this swap",(proxyBalance -
↪   USDC.token.balanceOf(address(swapperProxy))));

}
```

```
Ran 1 test for test/integration/Base/SwapperAndZapperAllTests.t.sol:SwapperAndZappe
↪   rAllTests8453
[PASS] test_integration_zapper_rescue_funds() (gas: 2630445)
Logs:
  How much do we have left in the contract: 99700000
  Amount stolen by attacker for this swap: 300000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 719.59ms (11.74ms CPU
↪   time)
```

Mitigation

Track the token-in balance before the swap and after the swap, return the excess balance immediately to msg.sender. OR Decode the Aggregator swap data to ensure that amount-in encoded is indeed the amount in transferred in.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/492

Issue M-6: Chainlink price decimals is wrongly is scaled

Source: https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/39

Found by

000000, 0xc0ffEE, santipu_

Summary

The price fetched from Chainlink Oracle is always scaled by 1e10, which can cause incorrect pricing for some assets

Root Cause

The function _getOnchainPrice() fetches price from Chainlink and then scale the price by 1e10 (implicitly assumed that price always has 8 decimals)

```
    function _getOnchainPrice(address asset_) internal view returns (uint256
 ↪  price_, uint256 timestamp_) {
        OracleModuleStructs.OracleData memory oracleData = _oracles[asset_];
        IChainlinkAggregatorV3 oracle = oracleData.onchainOracle.oracleContract;
        if (address(oracle) == address(0)) revert
 ↪  ICommonErrors.ZeroAddress("oracle");

        (, int256 price, , uint256 updatedAt, ) = oracle.latestRoundData();
        timestamp_ = updatedAt;
        // check Chainlink oracle price updated within `maxAge` time.
        if (block.timestamp > timestamp_ + oracleData.onchainOracle.maxAge)
            revert
 ↪  ICommonErrors.PriceStale(OracleModuleStructs.PriceSource.OnChain);

        if (price > 0) {
@>            price_ = uint256(price) * (10 ** 10); // convert Chainlink oracle
 ↪  decimals 8 -> 18
        } else {
            // Issue with onchain oracle indicates a serious problem
            revert
 ↪  ICommonErrors.PriceInvalid(OracleModuleStructs.PriceSource.OnChain);
        }
    }
```

This can cause asset pricing incorrectly. For example, the PEPE feed on Arb, or MEW feed, MOG feed on Base have 18 decimals

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

- Incorrect pricing for some assets

PoC

No response

Mitigation

Scale according to the feed's decimals

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/505

Issue M-7: Last long position to close will be stuck due to rounding up funding accrued by longs

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/59

Found by

KupiaSec, santipu_

Summary

The strict check on `updateGlobalPositionData` when closing the last long position on the protocol will prevent that position from being closed, causing a loss of funds for the owner of that long position.

Root Cause

In FlatcoinVault.sol::184, there is a check ensuring that when closing the last position on the protocol, the remaining margin must be lower than `1e6`.

```
// Close the last remaining position.
if (newMarginDepositedTotal > 1e6) revert MarginMismatchOnClose();

delete _globalPositions;
```

When the protocol has been running for some years, this check will fail because the funding fees are always rounded up in _accruedFundingTotalByLongs.

```
// To avoid rounding errors when subtracting individual position funding fees from
 ↪  the global `marginDepositedTotal` value,
// we add 1 wei to the total accrued funding by longs each time funding fees are
 ↪  settled provided that funding fees to be settled
// isn't 0.
return (accruedFundingTotal != 0) ? accruedFundingTotal + 1 : accruedFundingTotal;
```

If the funding fees are rounded up more than `1e6` times, then when the last position is closed, the remaining margin will be higher than `1e6`, causing the check to fail.

Internal pre-conditions

1. The protocol must be running for some years so that the fees are rounded up more than `1e6` times.

External pre-conditions

No response

Attack Path

1. Everytime an operation happens in the protocol, the funding fees are accrued (`settleFundingFees`). The function `_accruedFundingTotalByLongs` will be called, which will round up the fees earned by the longs, that will be added to the global margin.
2. If we assume that the average block time in Base is 2 seconds, and `settleFundingFees` may be called every 30 blocks (1 minute), then it will be called 1440 times a day. Before two years have passed, the funding fees would have been accrued more than `1e6` times, which will cause this issue when the last position is closed.

Impact

The last long position won't be able to be closed, causing a direct loss of funds for the owner of that position.

PoC

No response

Mitigation

To mitigate this issue, I recommend removing the strict check from `updateGlobalPositionData`:

```
    function updateGlobalPositionData(
        uint256 price_,
        int256 marginDelta_,
        int256 additionalSizeDelta_
    ) external onlyAuthorizedModule {
        // ...

        // Recompute the average entry price.
        if ((sizeOpenedTotal + additionalSizeDelta_) != 0) {
            // ...
        } else {
            // Close the last remaining position.
-           if (newMarginDepositedTotal > 1e6) revert MarginMismatchOnClose();

            delete _globalPositions;
        }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/507

Issue M-8: `ArbKeeperFee` will return incorrect inflated `keeperFee` for non-18 decimals tokens due to hard-coded constant `UNIT`.

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/84

Found by

0x37, Kirkeelee, santipu_, xiaoming90, zarkk01

Summary

The `ArbKeeperFee` is supposed to calculate and return the `keeperFee` that users will pay to keepers in the collateral token but it is doing it incorrectly for non-18 decimals tokens such as WBTC(with 8 decimals) which protocols expects to work with.

Root Cause

Let's see the `getKeeperFee` of `ArbKeeperFee` :

```
    function getKeeperFee(uint256 baseFee_) public view override returns (uint256
↪   keeperFeeCollateral_) {
        uint256 ethPrice18;
        uint256 collateralPrice;
        {
            uint256 timestamp;

            (, int256 ethPrice, , uint256 ethPriceupdatedAt, ) =
↪   _ethOracle.latestRoundData();

            if (block.timestamp >= ethPriceupdatedAt + _stalenessPeriod) revert
↪   ETHPriceStale();
            if (ethPrice <= 0) revert ETHPriceInvalid();

            ethPrice18 = uint256(ethPrice) * 1e10; // from 8 decimals to 18
            // NOTE: Currently the market asset and collateral asset are the same.
            // If this changes in the future, then the following line should fetch
↪   the collateral asset, not market asset.
            uint32 maxAge =
↪   _oracleModule.getOracleData(_assetToPayWith).onchainOracle.maxAge;
            (collateralPrice, timestamp) = _oracleModule.getPrice(_assetToPayWith);

            if (collateralPrice <= 0) revert
↪   ICommonErrors.PriceInvalid(OracleModuleStructs.PriceSource.OnChain);
```

```
            if (block.timestamp >= timestamp + maxAge)
                revert
↪   ICommonErrors.PriceStale(OracleModuleStructs.PriceSource.OnChain);
        }

        // fetch & define L1 gas base fee; incorporate overhead buffer
        /// @dev if the estimate is too low or high at the time of the L1 batch
↪   submission,
        /// the transaction will still be processed, but the arbitrum nitro
↪   mechanism will
        /// amortize the deficit/surplus over subsequent users of the chain
        /// (i.e. lowering/raising the L1 base fee for a period of time)
        uint256 l1BaseFee = IArbGasInfo(_gasPriceOracle).getL1BaseFeeEstimate();

        // (1) calculate total fee:
        //   -> total_fee = P * G
        // where:
        // (2) P is the L2 basefee
        //   -> P = L2 basefee
        // (3) G is gas limit that also accounts for L1 dimension
        //   -> G = L2 gas used + ( L1 calldata price * L1 calldata size) / (L2 gas
↪   price)
        uint256 costOfExecutionGrossEth = baseFee_ * (_gasUnitsL2 + ((l1BaseFee *
↪   _gasUnitsL1) / baseFee_));
        uint256 costOfExecutionGrossUSD =
↪   costOfExecutionGrossEth.mulDiv(ethPrice18, _UNIT); // fee priced in USD

@>        keeperFeeCollateral_ = (_keeperFeeUpperBound.min(costOfExecutionGrossUSD.⌋
↪   max(_keeperFeeLowerBound))).mulDiv(
            _UNIT,
            collateralPrice
        ); // fee priced in collateral
    }
```

Link to code

As we can see in the final line, the conversion from USD to collateral token is happening using `UNIT` constant which as we can see is equal to `1e18`:

```
abstract contract KeeperFeeBase is Ownable {
    /////////////////////////////////////////////
    //                  Errors                 //
    /////////////////////////////////////////////

    error ETHPriceStale();
    error ETHPriceInvalid();

    /////////////////////////////////////////////
    //                  State                  //
    ////////////////////////////////////////////////

    bytes32 public constant MODULE_KEY = FlatcoinModuleKeys._KEEPER_FEE_MODULE_KEY;

    address internal _gasPriceOracle; // Gas price oracles as deployed on L2s.
    IChainlinkAggregatorV3 internal _ethOracle; // ETH price for gas unit
 ↪  conversions
    IOracleModule internal _oracleModule; // for collateral asset pricing (the
 ↪  flatcoin market)

@>    uint256 internal constant _UNIT = 10 ** 18;
    uint256 internal _stalenessPeriod;

    // ...
}
```

Link to code

For this reason, the conversion is incorrect and it will not return the correct amount of `keeperFee` in `assetToPayWith` token.

For example, for WBTC with 8 decimals the result which where supposed to be in 8 decimals, will eventually be the same just in 18 decimals. The difference is huge and the result will be that EVERY order announcement will revert on **Arbitrum** network, unless the users pay this huge fees.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

The impact of this vulnerability is that the hardcoded _UNIT = 1e18 will cause the ArbKeeperFee contract to incorrectly calculate the keeperFee for tokens with decimals other than 18, such as WBTC (which has 8 decimals). This miscalculation will significantly inflates the fee amount, potentially making it 1e10 times larger than intended. As a result, transactions involving non-18 decimal tokens will either fail due to excessive fees or users will be forced to overpay. **Protocol expectes to work with WBTC as collateral token as it**

is public information on Discord channel. The result is that the protocol can **not** work on Arbitrum network while it is intended.

PoC

No response

Mitigation

Fix it as it is happening in the `OPKeeperFee` with the decimals of the `assetToPayWith` token.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/496

Issue M-9: `announcedOrders` that will increase `marginDeposited` are blocked, if funding fees have turned the global `marginDeposited` negative.

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/90

The protocol has acknowledged this issue.

Found by

0x37, KupiaSec, santipu_, zarkk01

Summary

`vault.checkGlobalMarginPositive()` during the execution of the orders will block all orders, even the ones that are supposed to increase the `marginDeposited`.

Root Cause

There is a crucial check that is operated on the protocol every time the funding fees are settled through `ControllerBase::settleFundingFees` and this is `vault.checkGlobalMarginPositive()`. Essentially, this is checking if the funding fees have "drained" the whole global `marginDepositedTotal` and turned it negative. If this is the case, all the orders will revert. Let's see the `OrderExecutionModule::executeOrder` :

```
function executeOrder(
    address account_,
    bytes[] calldata priceUpdateData_
)
    external
    payable
    nonReentrant
    whenNotPaused
    updatePythPrice(vault, msg.sender, priceUpdateData_)
    orderInvariantChecks(vault)
{
    DelayedOrderStructs.Order memory order = _getAnnouncedOrder(account_);

    if (order.orderType == DelayedOrderStructs.OrderType.None) revert
 ↪  OrderInvalid(account_);

    _prepareExecutionOrder(account_, order.executableAtTime);

    // Settle funding fees before executing any order.
    // This is to avoid error related to max caps or max skew reached when the
 ↪  market has been skewed to one side for a long time.
    IControllerModule(vault.moduleAddress(FlatcoinModuleKeys._CONTROLLER_MODULE_KEY ⌋
 ↪  )).settleFundingFees();

@>    vault.checkGlobalMarginPositive();

    // ...

    emit OrderExecuted({account: account_, orderType: order.orderType, keeperFee:
 ↪  order.keeperFee});
}
```

Link to code

And here is the `FilecoinVault::checkGlobalMarginPositive`:

```
function checkGlobalMarginPositive() public view {
    int256 globalMarginDepositedTotal = _globalPositions.marginDepositedTotal;

    if (globalMarginDepositedTotal < 0) revert InsufficientGlobalMargin();
}
```

Link to code

However, this shouldn't be the case always. If the order is a `DelayedOrderStructs.OrderType.LeverageAdjust` order and will increase the corresponding margin and thus the global `marginDeposited`, this order must not revert since it will bring the protocol and traders back to more healthy levels. By blocking **any** order when the global `marginDeposited` turns negative due to the funding fees, the protocol gets essentially bricked and no trader will be able to increase his marging

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. LP provide liquidity and trader opens leverage position.
2. Global `marginDeposited` gets negative from funding fees.
3. Every order is reverted, even though the `LeverageAdjust` ones that will increase the `marginDeposited` must be passing

Impact

The impact of this vulnerability is that when the global marginDeposited turns negative due to funding fees, all orders are blocked, including those meant to increase the margin. This prevents traders from adding more funds to their positions, which could help stabilize the system. As a result, the protocol becomes stuck, and no actions can be taken to recover. This can harm both traders and the platform's overall health.

PoC

No response

Mitigation

Consider not blocking the `announcedOrders` that will increase the `marginDeposited`.

Issue M-10: There is a significant rounding error when WBTC is used as collateral.

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/98

Found by

KupiaSec

Summary

The collateral price of WBTC is about 1e18*1e5=1e23, because the price decimal of `OracleModule` is 18. So, the initial `collateralPerShare` will be about 1e18 *1e8/1e23 = 1e3. Therefore, if the collateral is WBTC, the rounding error will be up to about 0.1%.

Root Cause

The collateral price of WBTC is approximately `1e18 * 1e5 = 1e23`, since the price decimal of the OracleModule is 18. Consequently, the initial collateralPerShare will be about `1e18 * 1e8 / 1e23 = 1e3`. This indicates that if the collateral is WBTC, the rounding error could be as much as `1/1000`, or `0.1%`.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/StableModule.sol#L217-L237

```
    function stableCollateralPerShare(
        uint32 maxAge_,
        bool priceDiffCheck_
    ) public view returns (uint256 collateralPerShare_) {
        uint256 totalSupply = totalSupply();

        if (totalSupply > 0) {
            uint256 stableBalance = stableCollateralTotalAfterSettlement({
                maxAge_: maxAge_,
                priceDiffCheck_: priceDiffCheck_
            });
            collateralPerShare_ = (stableBalance * (10 ** decimals())) /
 ↪  totalSupply;
        } else {
            IERC20Metadata collateral = vault.collateral();
            (uint256 collateralPrice, ) =
 ↪  IOracleModule(vault.moduleAddress(FlatcoinModuleKeys._ORACLE_MODULE_KEY))
                .getPrice({asset: address(collateral), maxAge: maxAge_,
 ↪  priceDiffCheck: priceDiffCheck_});
```

```
            // no shares have been minted yet
            collateralPerShare_ = (1e18 * (10 ** collateral.decimals())) /
    collateralPrice;
        }
    }
```

As a result, depositors may receive different amounts of shares (with discrepancies of up to approximately 0.1%) in nearly identical situations due to rounding errors. Similarly, withdrawers may also receive varying amounts of WBTC (up to about 0.1% discrepancy) under nearly identical pool conditions.

Internal pre-conditions

none

External pre-conditions

none

Attack Path

none

Impact

Significant calculation error.

PoC

none

Mitigation

The larger decimal should be used for `collateralPerShare`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/518

Issue M-11: Some invariant checks will not work when the collateral is WBTC.

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/99

Found by

KupiaSec

Summary

`InvariantChecks.sol` implements several invariant checks, but it applies a uniform threshold of 1e6 for rounding errors, irrespective of the collateral type. This approach is inadequate for certain collaterals, such as WBTC, where the checks may not function effectively.

Root Cause

The contest README says that:

> Q: What properties/invariants do you want to hold even if breaking them has a low/unknown impact?
>
> Any set of transactions which can break the invariants/properties as mentioned in the InvariantChecks.sol contract file without reverting.

At L126, there is a invariant check that collateral balance changes should match tracked collateral changes.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/StableModule.sol#L122-L128

```
function _collateralNetBalanceRemainsUnchanged(int256 netBefore_, int256 netAfter_)
 ↪  private pure {
    // Note: +1e6 to account for rounding errors.
    // This means we are ok with a small margin of error such that netAfter - 1e6
 ↪  <= netBefore <= netAfter.
    if (netBefore_ > netAfter_ || netAfter_ > netBefore_ + 1e6)
        revert InvariantChecks.InvariantViolation("collateralNet2");
}
```

There is another check for change of `stableCollateralPerShare` involved in liquidation invariant checks at L177.

However, in the above two invariant checks, `1e6` is used as a limit, even when the collateral is WBTC.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/StableModule.sol#L142-L180

```
            if (
131:            stableCollateralPerShareAfter < stableCollateralPerShareBefore -
 ↪  1e6 ||
                stableCollateralPerShareAfter > stableCollateralPerShareBefore + 1e6
            ) revert PriceImpactDuringWithdraw();
```

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/StableModule.sol#L70-L102

```
leverageModule.checkLeverageCriteria(primaryMargin, primarySize);
leverageModule.checkLeverageCriteria(newPositionMargin, newPositionSize);
```

The current price of 1 WBTC is about 1e5USD, so the initial `stableCollateralPerShare` of WBTC is about `(1e18* 1e8)/(1e18*1e5)=1e3` So, the check at L177 will always pass when the collateral is WBTC. This vulnerability shows the invalid invariant check when WBTC is collateral. And, at L126, the limit `1e6` means 0.01 WBTC, which is about 1000 USD. As a result, the above two checks will not work well, when the collateral is WBTC.

Similar vulnerability also occurs in `FlatcoinVault.sol`, so the check also does not work when the collateral is WBTC.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L183-L186

```
            // Close the last remaining position.
@>          if (newMarginDepositedTotal > 1e6) revert MarginMismatchOnClose();

            delete _globalPositions;
```

Internal pre-conditions

The collateral is WBTC.

External pre-conditions

none

Attack Path

none

Impact

Some invariant check does not work.

PoC

none

Mitigation

Different thresholds should be used for different collaterals.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/519

Issue M-12: Order announcements can be DoSed.

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/100

Found by

000000, 0x37, KupiaSec, santipu_

Summary

An honest user wouldn't be able to manage his position properly because an attacker announces an order with the `receiver` as the user.

Root Cause

In the new version, `OrderAnnouncementModule::announceStableDepositFor()` and `OrderAnnouncementModule::announceLeverageOpenFor()` were added and these functions are used when users announce orders through the `FlatZapper` contract for StableDeposit and LeverageOpen.

```
File: OrderAnnouncementModule.sol
    function announceStableDepositFor(
        uint256 depositAmount_,
        uint256 minAmountOut_,
        uint256 keeperFee_,
        address receiver_
    ) public whenNotPaused {}

    function announceLeverageOpenFor(
        uint256 margin_,
        uint256 additionalSize_,
        uint256 maxFillPrice_,
        uint256 stopLossPrice_,
        uint256 profitTakePrice_,
        uint256 keeperFee_,
        address receiver_
    ) public whenNotPaused {}
```

But an attacker can call these functions directly and block other user's operations for DoS or his profit.

Scenario 1

1. A user should approve their balance to `OrderAnnouncementModule` for order announcement, then the approve event will be emitted.

2. An attacker can catch the event and call `announceStableDepositFor()` with `minDepositAmountUSD + keeperFee`.
3. Then, the legitimate announcement will revert, because an announced call cannot be canceled for `minExecutabilityAge + maxExecutabilityAge` seconds.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/OrderExecutableModule.sol#L165-L224

```
function test_adjust_position_margin_increase_will_revert_an_healthy_position()
↪  public {
     uint256 aliceCollateralBalanceBefore = collateralAsset.balanceOf(alice);
     uint256 stableDeposit = 100e18;
     uint256 collateralPrice = 1000e8;
     uint256 keeperFee = mockKeeperFee.getKeeperFee();

     announceAndExecuteDeposit({
         traderAccount: alice,
         keeperAccount: keeper,
         depositAmount: stableDeposit,
         oraclePrice: collateralPrice,
         keeperFeeAmount: keeperFee
     });

     // 10 ETH margin, 30 ETH size (4x)
     uint256 tokenId = announceAndExecuteLeverageOpen({
         traderAccount: alice,
         keeperAccount: keeper,
         margin: 10e18,
         additionalSize: 30e18,
         oraclePrice: collateralPrice,
         keeperFeeAmount: keeperFee
     });





   uint256 liqPrice = viewer.liquidationPrice(0);



 LeverageModuleStructs.Position memory position = vaultProxy.getPosition(tokenId);
 LeverageModuleStructs.PositionSummary memory positionSummary
↪  =leverageModProxy.getPositionSummary(position, (liqPrice));

 emit log_named_int("Profitloss", positionSummary.profitLoss);
 emit log_named_int("accrue", positionSummary.accruedFunding);
 emit log_named_int("Margin at liquidation", positionSummary.marginAfterSettlement);
 emit log_named_uint("Liquidation price", liqPrice);


uint premargin =  liquidationModProxy.getLiquidationMargin(31e18, liqPrice);
 emit log_named_uint("liquidation margin at liquidation price", premargin);



 LeverageModuleStructs.Position memory position2 = vaultProxy.getPosition(tokenId);
 LeverageModuleStructs.PositionSummary memory positionSummary2
↪   =leverageModProxy.getPositionSummary(position2, ((liqPrice + 1e18) - ((liqPrice
↪   + 1e18)/1000)));

 emit log_named_int("Profitloss2", positionSummary2.profitLoss);
 emit log_named_int("accrue2", positionSummary2.accruedFunding);
 emit log_named_int("Current price Margin", positionSummary2.marginAfterSettlement);
 emit log_named_uint("current Price Now", ((liqPrice + 1e18) - ((liqPrice +
↪   1e18)/1000)));



   uint256 newCollateralPrice = ((liqPrice + 1e18) - ((liqPrice + 1e18)/1000));


      setCollateralPrice(newCollateralPrice/1e10);







     uint256 keeperFee2 = mockKeeperFee.getKeeperFee();


   vm.startPrank(alice);

     collateralAsset.approve(address(orderAnnouncementModProxy), 6e18);





     emit log_named_uint("fill price", liqPrice+1e18);
```

```
      uint premargin5 =  liquidationModProxy.getLiquidationMargin(31e18,
 ↪  (liqPrice+1e18));
      emit log_named_uint("Liquidation Margin calculated during annoucement using
 ↪  fillprice", premargin5);




      LeverageModuleStructs.Position memory position5 =
 ↪  vaultProxy.getPosition(tokenId);
     LeverageModuleStructs.PositionSummary memory positionSummary5
 ↪  =leverageModProxy.getPositionSummary(position5,liqPrice+1e18);

       emit log_named_int("Profitloss5", positionSummary5.profitLoss);
      emit log_named_int("accrue5", positionSummary5.accruedFunding);
       emit log_named_int("Actual Margin at Fill Price",
 ↪  positionSummary5.marginAfterSettlement);

       assert(premargin5 > uint(positionSummary2.marginAfterSettlement));

        assert(premargin5 > uint(positionSummary2.marginAfterSettlement));
        assert(uint(positionSummary5.marginAfterSettlement) > premargin5);

 }
```

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/OrderAnnouncementModule.sol#L589

```
    function test_address_zero_blocks_orders() public {
        // We open an order for address(0) to set the trap
        announceStableDepositFor({
            traderAccount: keeper,
            receiver: address(0),
            depositAmount: 1e18,
            keeperFeeAmount: 0
        });

        // A normal user deposits
        announceAndExecuteDeposit({
            traderAccount: bob,
            keeperAccount: keeper,
            depositAmount: 100e18,
            oraclePrice: 1000e8,
            keeperFeeAmount: 0
        });

        // We now try to open other leverage positions
        // It reverts here with `OrderExists(1)`
        announceAndExecuteLeverageOpen({
            traderAccount: alice,
            keeperAccount: keeper,
            margin: 60e18,
            additionalSize: 100e18,
            oraclePrice: 1000e8,
            keeperFeeAmount: 0
        });
    }
```

4. Creating an order is time-sensitive, so DoS of order announcement can lead to potential loss of fund to users.

Scenario 2

1. A user has a large position which is liquidatable if the collateral price drops a little more.
2. So the user is going to close or adjust this position.
3. An attacker who wants to charge a liquidation fee decides to block the user's operation.
4. So the attacker continues to call `announceStableDepositFor()` or `announceLeverageOpenFor()` for the user with a minimum fund whenever it's possible.
5. For the same reason as scenario 1, it will block the user's operation for `minExecutabilityAge + maxExecutabilityAge` seconds if it's not executed by keepers, or `minExecutabilityAge` seconds at least if executed.
6. Finally, the risky position is liquidated by the attacker and he will charge the liquidation fee.

Internal pre-conditions

none

External pre-conditions

none

Attack Path

none

Impact

DoS of legitimate order announcement.

PoC

none

Mitigation

`announceStableDepositFor()` and `announceLeverageOpenFor()` should be called from the `FlatZapper` contract only.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/504

Issue M-13: In the executeWithdraw() function, the protocol fee is not considered when it checks the system's skew

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/102

Found by

000000, 0xc0ffEE, Afriaudit, Bigsam, KupiaSec, newspacexyz, santipu_, zarkk01

Summary

When depositors withdraw collateral, the system checks if the skew exceeds the `skewFractionMax` after the withdrawal. However, in the executeWithdraw() function, the protocol fee is not considered during this skew check. As a result, the protocol may incorrectly permit a stable withdrawal, even when the execution is fundamentally unsafe. This violates the skew mechanism of the protocol, compromising the overall system's safety.

Root Cause

At StableModule.sol#139, it checks if the system's skew exceeds the `skewFractionMax`.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/StableModule.sol#L139

```
/// @notice Swap tokens using the given `swapStruct_`.
    /// @dev Only supports SINGLE_IN_SINGLE_OUT and MULTI_IN_SINGLE_OUT swap types.
    /// @param swapStruct_ The struct containing all the data required to process
↪   the swap(s).
    function swap(SwapperStructs.InOutData calldata swapStruct_) external payable {
        uint256 destAmountBefore =
↪   swapStruct_.destData.destToken.balanceOf(address(this));

        // Transfer all the `srcTokens` to this contract.
        _transferFromCaller(msg.sender, swapStruct_.srcData);

        // Process swaps based on `srcData` array.
        // The first loop iterates over the `srcData` array. The number of
↪   iterations is equal to the number of transfer methods used in the swap.
        // For example if the swap uses `TokenTransferMethod.ALLOWANCE` for all
↪   `srcTokens`, then the outer loop will iterate only once.
        // If the swap uses `TransferMethod.ALLOWANCE` for the first x `srcTokens`
↪   and `TransferMethod.PERMIT` for the next y `srcTokens`,
        // then the outer loop will iterate twice.
        for (uint256 i; i < swapStruct_.srcData.length; ++i) {
            // The second loop iterates over the `srcTokens` array in which the
↪   `srcTokens` are transferred and swapped using the same token transfer method.
            for (uint256 j; j < swapStruct_.srcData[i].srcTokenSwapDetails.length;
↪   ++j) {
                _processSwap({srcTokenSwapDetails_:
↪   swapStruct_.srcData[i].srcTokenSwapDetails[j]});
            }
        }

        // Check that we got enough of each `destToken` after processing and
↪   transfer them to the caller.
        // Note that we don't consider the current `destToken` balance of this
↪   contract as the received amount
        // as the amount can be more than the actual received amount due to someone
↪   else transferring tokens to this contract.
        // The following approach gives us the ability to rescue funds from this
↪   contract.
        uint256 destAmountReceived =
↪   swapStruct_.destData.destToken.balanceOf(address(this)) - destAmountBefore;

        if (destAmountReceived < swapStruct_.destData.minDestAmount)
            revert InsufficientAmountReceived(
                swapStruct_.destData.destToken,
                destAmountReceived,
                swapStruct_.destData.minDestAmount
            );

        swapStruct_.destData.destToken.safeTransfer(msg.sender, destAmountReceived);
    }
```

As indicated in the contest README, the protocol fee should be deducted from the withdrawal fee.

> Added protocol fees component to take a portion of fees earned (from trading and LP withdrawals).

So, at L139 the `stableCollateralChange` variable should be `withdrawFee - FeeManager(address(vault)).getProtocolFee(withdrawFee)`.

Internal pre-conditions

`protocolFeePercentage` should be greater than `0`.

External pre-conditions

N/A

Impact

The protocol may incorrectly permit a stable withdrawal, even when the execution is fundamentally unsafe. This also violates the skew mechanism of the protocol, compromising the overall system's safety.

PoC

When depositors withdraw the collateral, the total stable collateral of the vault is updated.

```
123:     vault.updateStableCollateralTotal(-int256(amountOut_));
```

The `withdrawFee_` is calculated and the checkSkewMax() function is called to ensure that the system will not be too skewed towards longs.

```
  /// @inheritdoc IUniversalRouter
    function execute(bytes calldata commands, bytes[] calldata inputs) public
↪   payable isNotLocked {
        bool success;
        bytes memory output;
        uint256 numCommands = commands.length;
        if (inputs.length != numCommands) revert LengthMismatch();

        // loop through all given commands, execute them and pass along outputs as
↪   defined
        for (uint256 commandIndex = 0; commandIndex < numCommands;) {
            bytes1 command = commands[commandIndex];

            bytes memory input = inputs[commandIndex];

            (success, output) = dispatch(command, input);

            if (!success && successRequired(command)) {
                revert ExecutionFailed({commandIndex: commandIndex, message:
↪   output});
            }

            unchecked {
                commandIndex++;
            }
        }
```

At the end of the execution, the total stable collateral of the vault is updated again with `withdrawFee - protocolFee`, the protocol fee is transferred to the protocol fee receipient.

```
282:         vault.updateStableCollateralTotal(int256(withdrawFee - protocolFee));
↪   // pay the withdrawal fee to stable LPs
```

```
283:         vault.sendCollateral({to:
↪ FeeManager(address(vault)).protocolFeeRecipient(), amount: protocolFee}); //
↪ pay the protocol fee
```

This means that the protocol fee is not included in the vault; however, it is added when checking the skew maximum. As a result, this could incorrectly permit withdrawals that should not occur.

Consider the following scenario:

1. `skewFractionMax` is `120%` and `stableWithdrawFee` is 1%, `protocolFee` is `10%`.
2. Alice deposits `10000` collateral and Bob opens a leverage position with size `10000`.
3. At the moment, there is `10000` collaterals in the vault, skew fraction is `100%`.
4. Alice tries to withdraw `1683` collaterals, withdrawFee is `16.83`, protocolfee is `1.683`. After withdrawal, it is expected that there is `10000 - 1683 + 16.83 - 1.683 = 8332.147` stable collaterals in the Vault, so `skewFraction` should be `100 * 10000 / 8332.147 = 120.017%`, which is greater than `skewFractionMax`.
5. However, the withdrawal will actually succed because when protocol checks skew max, protocolfee is ignored and the `skewFraction` turns out to be `100 * 10000 / (10000 - 1683 + 16.83) = 100 * 10000 / 8333.83 = 119.993%`, which is lower than `skewFractionMax`.

Mitigation

It is recommended to change the code as follows.

```
function test_integration_zapper_rescue_funds() public {
   // Ensure no active prank before starting a new one
   vm.stopPrank();

   // Initialize the source token array with USDC
   Token[] memory srcTokens = TokenArrayBuilder.fill(1, USDC);
   Token memory destToken = rETH;

   // Start impersonating 'bob'
   vm.startPrank(bob);

   // Save Bob's USDC balance before the transfer
   uint256 bobBalanceBefore = USDC.token.balanceOf(bob);

   // Bob transfers USDC directly to the Swapper Proxy
   uint256 transferAmount = 100e6; // Amount in USDC (assuming 6 decimals) //
↪  donation to to if residue tokens can be stolen
   require(USDC.token.balanceOf(bob) >= transferAmount, "Bob has insufficient USDC
↪  balance");
   USDC.token.transfer(address(swapperProxy), transferAmount);
```

```
    // Validate that the transfer was successful
    uint256 proxyBalance = USDC.token.balanceOf(address(swapperProxy));
    require(proxyBalance >= transferAmount, "USDC transfer to Swapper Proxy
↪   failed");

     // Stop impersonating 'bob'
    vm.stopPrank();

    // Run all aggregator tests with the specified parameters
    runAllAggregatorTests({
        srcTokens: srcTokens,
        destToken: destToken,
        transferMethod: SwapperStructs.TransferMethod.ALLOWANCE
    });

    emit log_named_uint("How much do we have left in the
↪   contract",USDC.token.balanceOf(address(swapperProxy)));
     emit log_named_uint("Amount stolen by attacker for this swap",(proxyBalance -
↪   USDC.token.balanceOf(address(swapperProxy))));

}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/499

Issue M-14: `marginDepositedTotal` could be less than the sum of `marginDeposited`(Invariant breaking)

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/105

Found by

KupiaSec

Summary

There is `+1` in ControllerBase.sol#L347 to ensure that `GlobalMargin` remains greater than sum of `margins`. However, this is insufficiant, because there are several calculations for traders round down.

Root Cause

```
Ran 1 test for test/integration/Base/SwapperAndZapperAllTests.t.sol:SwapperAndZappe
↪  rAllTests8453
[PASS] test_integration_zapper_rescue_funds() (gas: 2630445)
Logs:
  How much do we have left in the contract: 99700000
  Amount stolen by attacker for this swap: 300000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 719.59ms (11.74ms CPU
↪  time)
```

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/tree/main/flatcoin-v1/src/libraries/DecimalMath.sol#L35 In a single block, the `marginDepositedTotal` is increassed by 1 only once, while several calculations(for increasing position) for traders round down when `stableCollateralTotal > sizeOpenedTotal`. As a result, the `marginDepositedTotal_aftersettlement` could be less than sum of margins.

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

N/A

Impact

Invariant breaking and the last trader can't close his/her position.

PoC

```
    // To avoid rounding errors when subtracting individual position funding fees
↪   from the global `marginDepositedTotal` value,
    // we add 1 wei to the total accrued funding by longs each time funding fees
↪   are settled provided that funding fees to be settled
    // isn't 0.
    return (accruedFundingTotal != 0) ? accruedFundingTotal + 1 :
↪   accruedFundingTotal;
```

When `x * y < 0`, `x * y / UNIT` is rounded up.

```
ControllerBase.sol
217:function accruedFunding(
        LeverageModuleStructs.Position memory position_
    ) public view virtual returns (int256 accruedFunding_) {
        int256 net = _netFundingPerUnit(position_.entryCumulativeFunding);

        return int256(position_.additionalSize)._multiplyDecimal(net);
    }
```

When `stableCollateralTotal < _globalPositions.sizeOpenedTotal`, `net` becomes negative. At this time, if `position_.additionalSize > 0`, `accruedFunding_`(return value) is rounded up.

```
LeverageModule.sol
405:function getPositionSummary(
        LeverageModuleStructs.Position memory position_,
        uint256 price_
    ) public view returns (LeverageModuleStructs.PositionSummary memory
↪   positionSummary_) {
        ...
        int256 accruedFundingOfPosition = perpController.accruedFunding(position_);

        return
            LeverageModuleStructs.PositionSummary({
                profitLoss: profitLossOfPosition,
419:            accruedFunding: accruedFundingOfPosition,
                marginAfterSettlement: int256(position_.marginDeposited) +
                    profitLossOfPosition +
                    accruedFundingOfPosition
            });
    }
299:function executeClose(
        DelayedOrderStructs.Order calldata order_
    ) external onlyAuthorizedModule returns (uint256 marginAfterPositionClose_) {
        ...
        uint256 totalFee;
        int256 settledMargin;
        LeverageModuleStructs.PositionSummary memory positionSummary;
        {
            positionSummary = getPositionSummary(position, exitPrice);
            ...
            vault.updateGlobalPositionData({
                price: position.averagePrice,
339:            marginDelta: -(int256(position.marginDeposited) +
 ↪  positionSummary.accruedFunding),
                additionalSizeDelta: -int256(position.additionalSize)
            });
            ...
        }
        ...
    }
```

The `marginDelta`(L339) is rounded-down value.(Because of the - sign)

```
function executeOrder(
    address account_,
    bytes[] calldata priceUpdateData_
)
    external
    payable
    nonReentrant
    whenNotPaused
    updatePythPrice(vault, msg.sender, priceUpdateData_)
    orderInvariantChecks(vault)
{
    DelayedOrderStructs.Order memory order = _getAnnouncedOrder(account_);

    if (order.orderType == DelayedOrderStructs.OrderType.None) revert
 ↪  OrderInvalid(account_);

    _prepareExecutionOrder(account_, order.executableAtTime);

    // Settle funding fees before executing any order.
    // This is to avoid error related to max caps or max skew reached when the
 ↪  market has been skewed to one side for a long time.
    IControllerModule(vault.moduleAddress(FlatcoinModuleKeys._CONTROLLER_MODULE_KEY
 ↪  )).settleFundingFees();
@>    vault.checkGlobalMarginPositive();

    // ...

    emit OrderExecuted({account: account_, orderType: order.orderType, keeperFee: order.keeperFee});
}
```

The `marginDepositedTotal`(L178) is rounded-down value. As a result, when subtracting individual position's `funding fees` from the global `marginDepositedTotal` value, the global value is decreased little by little.

```
    function _collateralNetBalanceRemainsUnchanged(int256 netBefore_, int256 netAfter_) private pure {
        // Note: +1e6 to account for rounding errors.
        // This means we are ok with a small margin of error such that netAfter - 1e6 <= netBefore <= netAfter.
        if (netBefore_ > netAfter_ || netAfter_ > netBefore_ + 1e6)
            revert InvariantChecks.InvariantViolation("collateralNet2");
    }
```

When all traders close their position, `MarginDepositedTotal` is deleted instead of going below zero.

```
File: OrderAnnouncementModule.sol
    function announceStableDepositFor(
        uint256 depositAmount_,
        uint256 minAmountOut_,
        uint256 keeperFee_,
        address receiver_
    ) public whenNotPaused {}

    function announceLeverageOpenFor(
        uint256 margin_,
        uint256 additionalSize_,
        uint256 maxFillPrice_,
        uint256 stopLossPrice_,
        uint256 profitTakePrice_,
        uint256 keeperFee_,
        address receiver_
    ) public whenNotPaused {}
```

At this point, the invariant is broken. Because the `collateralNet` is decreased. For example: `stableCollateralTotal = 1e18`, `marginDepositedTotal = 0.1e18 - 3`, last `trader's margin = 0.1e18`. `collateralBalance = 1.1e18`, `trackedCollateral = 1.1e18 -3`, `netCollateral = 3`. After close last trader's position, `collateralBalance = 1e18`, `trackedCollateral = 1e18`, `netcollateral = 0`. Because the NetBalanceRemain check(InvariantChecks.sol::L59), this last trader's closing is revered. Or For example:

`stableCollateralTotal = 1e18`, `marginDepositedTotal = 0.1e18 - 3`, `last traders margin = 0.1e18`. `collateralBalance = 1.1e18-3`, `trackedCollateral = 1.1e18 -3`, `netCollateral = 0`. After close last trader's position, `collateralBalance = 1e18-3`, `trackedCollateral = 1e18`, `netcollateral = -3`. Because the check in L112, this last trader's closing is reverted.

As a result, the last trader can't close his/her position.

Mitigation

It is more important that individual values are rounded down (when subtracting individual position funding fees from the global `marginDepositedTotal` value) than that the global value be increased by 1. If the code is to be written as commented, it should be as follows:

```
DecimalMath.sol
     * @dev A unit factor is divided out after the product of x and y is evaluated,
     * so that product must be less than 2**256. As this is an integer division,
     * the internal division always rounds down. This helps save on gas. Rounding
     * is more expensive on gas.
     */
    function _multiplyDecimal(int256 x, int256 y) internal pure returns (int256) {
        /* Divide by UNIT to remove the extra factor introduced by the product. */
-       return (x * y) / UNIT;
+       return (x * y) / UNIT - ((x * y % UNIT != 0 && x * y < 0) ? int256(1) :
 ↪  int256(0));
    }
```

Or

```
    function _prepareAnnouncementOrder(
        uint256 keeperFee_,
        address receiver_
 ) internal returns (uint64 executableAtTime_) {
        _preAnnouncementChores();

        if (keeperFee_ < IKeeperFee(vault.moduleAddress(FlatcoinModuleKeys._KEEPER_
 ↪  FEE_MODULE_KEY)).getKeeperFee())
            revert ICommonErrors.InvalidFee(keeperFee_);

        // If the user has an existing pending order that expired, then cancel it.
@>      IOrderExecutionModule(vault.moduleAddress(FlatcoinModuleKeys._ORDER_EXECUTI
 ↪  ON_MODULE_KEY)).cancelExistingOrder(
 receiver_
 );

@>      executableAtTime_ = uint64(block.timestamp + minExecutabilityAge);
 }
```

Test Code

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/tree/main/flatcoin-v1/test/unit/Delayed-Order/DelayedOrder.t.sol#L500 Changed this function to above code.

```
139: vault.checkSkewMax({sizeChange: 0, stableCollateralChange:
↪  int256(withdrawFee_)});
```

forge test DelayedOrder.t.sol

[FAIL: InvariantViolation("collateralNet1")] test_revert_announce_orders_when_global_margin_negative() (gas: 8672861) Logs: Trader: 4 Trader's MarginAfterSettlement is: 1699999999108434608 MarginDepositedTotal_Before is : 55200000002107445969 MarginDepositedTotal_After is : 53499999971941912642 SizeOpenedTotal is : 53500000000000000000 Trader: 3 Trader's MarginAfterSettlement is: 1699999999108434608 MarginDepositedTotal_Before is : 53499999971941912642 MarginDepositedTotal_After is : 51799999972833478034 SizeOpenedTotal is : 51800000000000000000 Trader: 2 Trader's MarginAfterSettlement is: 1699999999108434608 MarginDepositedTotal_Before is : 51799999972833478034 MarginDepositedTotal_After is : 50099999973725043426 SizeOpenedTotal is : 50100000000000000000 Trader: 1 Trader's MarginAfterSettlement is: 50099999973725043427 MarginDepositedTotal_Before is : 50099999973725043426

You can check that there are no FAIL by using the above mitigated functions.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/dhedge/flatcoin-v1/pull/506

Issue M-15: `collateralNet` could be decreased (Invariant breaking)

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/106

Found by

KupiaSec, vinica_boy

Summary

The `collateralNet` in the `FlatcoinVault` is intended to be non-decreasing; however, computational errors can lead to violations of this invariant.

Root Cause

The issue arises from the `FlatcoinVault.sol::updateGlobalPositionData()#L186`, where `_globalPositions` is deleted. This can lead to discrepancies in the `collateralNet`, especially.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/tree/main/flatcoin-v1/src/FlatcoinVault.sol#L186

```
FlatcoinVault.sol
    function updateGlobalPositionData(
        uint256 price_,
        int256 marginDelta_,
        int256 additionalSizeDelta_
    ) external onlyAuthorizedModule {
        // Note that technically, even the funding fees should be accounted for
↪   when computing the margin deposited total.
        // However, since the funding fees are settled at the same time as the
↪   global position data is updated,
        // we can ignore the funding fees here.
        int256 newMarginDepositedTotal = _globalPositions.marginDepositedTotal +
↪   marginDelta_;

        int256 averageEntryPrice = int256(_globalPositions.averagePrice);
        int256 sizeOpenedTotal = int256(_globalPositions.sizeOpenedTotal);

        // Recompute the average entry price.
        if ((sizeOpenedTotal + additionalSizeDelta_) != 0) {
            int256 newAverageEntryPrice = ((averageEntryPrice * sizeOpenedTotal) +
                (int256(price_) * additionalSizeDelta_)) / (sizeOpenedTotal +
↪   additionalSizeDelta_);
```

```
            _globalPositions = FlatcoinVaultStructs.GlobalPositions({
                marginDepositedTotal: newMarginDepositedTotal,
                sizeOpenedTotal: (int256(_globalPositions.sizeOpenedTotal) +
↪   additionalSizeDelta_).toUint256(),
                averagePrice: uint256(newAverageEntryPrice)
            });
        } else {
            // Close the last remaining position.
            if (newMarginDepositedTotal > 1e6) revert MarginMismatchOnClose();

186:        delete _globalPositions;
        }
    }
InvariantChecks.sol
107:function _getCollateralNet(IFlatcoinVault vault_) private view returns (int256
↪   netCollateral_) {
        int256 collateralBalance =
↪   int256(vault_.collateral().balanceOf(address(vault_)));
        int256 trackedCollateral = int256(vault_.stableCollateralTotal()) +
            vault_.getGlobalPositions().marginDepositedTotal;

        if (collateralBalance < trackedCollateral) revert
↪   InvariantChecks.InvariantViolation("collateralNet1");

        return collateralBalance - trackedCollateral;
    }
123:function _collateralNetBalanceRemainsUnchanged(int256 netBefore_, int256
↪   netAfter_) private pure {
        // Note: +1e6 to account for rounding errors.
        // This means we are ok with a small margin of error such that netAfter -
↪   1e6 <= netBefore <= netAfter.
        if (netBefore_ > netAfter_ || netAfter_ > netBefore_ + 1e6)
            revert InvariantChecks.InvariantViolation("collateralNet2");
    }
```

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

N/A

PoC

The formula for `netcollateral` is: `netCollateral = trackedCollateral - collateralBalance`, where `trackedCollateral = stableCollateralTotal + marginDepositedTotal`. If `marginDepositedTotal` decreases, `netCollateral` also decreases. The `_globalPositions` are deleted in the `FlatcoinVault.sol::updateGlobalPositionData()` function. It is difficult for `marginDepositedTotal` to maintain the correct value due to rounding errors. As a result, `marginDepositedTotal` may be slightly greater or slightly less than expected. When the last user closes their long position, if `marginDepositedTotal` is greater than 0, `netCollateral` decreases, and the user's transaction is reverted.

Impact

This can lead an invariant violation, resulting a situations where the last trader is unable to close their position.

Mitigation

```
ControllerBase.sol
217:function accruedFunding(
        LeverageModuleStructs.Position memory position_
    ) public view virtual returns (int256 accruedFunding_) {
        int256 net = _netFundingPerUnit(position_.entryCumulativeFunding);

        return int256(position_.additionalSize)._multiplyDecimal(net);
    }
```

Test Code

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/tree/main/flatcoin-v1/test/unit/Delayed-Order/DelayedOrder.t.sol#L500 Changed this function to above code.

```
LeverageModule.sol
405:function getPositionSummary(
        LeverageModuleStructs.Position memory position_,
        uint256 price_
    ) public view returns (LeverageModuleStructs.PositionSummary memory
↪   positionSummary_) {
        ...
        int256 accruedFundingOfPosition = perpController.accruedFunding(position_);

        return
            LeverageModuleStructs.PositionSummary({
                profitLoss: profitLossOfPosition,
419:            accruedFunding: accruedFundingOfPosition,
                marginAfterSettlement: int256(position_.marginDeposited) +
                    profitLossOfPosition +
                    accruedFundingOfPosition
            });
    }
299:function executeClose(
        DelayedOrderStructs.Order calldata order_
    ) external onlyAuthorizedModule returns (uint256 marginAfterPositionClose_) {
        ...
        uint256 totalFee;
        int256 settledMargin;
        LeverageModuleStructs.PositionSummary memory positionSummary;
        {
            positionSummary = getPositionSummary(position, exitPrice);
            ...
            vault.updateGlobalPositionData({
                price: position.averagePrice,
339:            marginDelta: -(int256(position.marginDeposited) +
↪   positionSummary.accruedFunding),
                additionalSizeDelta: -int256(position.additionalSize)
            });
            ...
        }
        ...
    }
```

forge test DelayedOrder.t.sol

Result: [FAIL: InvariantViolation(”collateralNet1”)] test_revert_announce_orders_when_global_margin_negative() (gas: 12617398)

You can check that there are no FAIL by using the above mitigated `updateGlobalPositionData()` function.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/dhedge/flatcoin-v1/pull/501

Issue M-16: Share Decreasing(Invariant breaking)

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/107

Found by

KupiaSec

Summary

`Rounding up` after `rounding down` is not equivalent to `rounding up`. Beause if the fraction is 0.09, this value is rounded down. This will ensure the traders get extra margin after settlement of the position and break the `share`'s invarinat.

Root Cause

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/tree/main/flatcoin-v1/src/LeverageModule.sol#L229

```
InvariantChecks.sol
    modifier orderInvariantChecks(IFlatcoinVault vault_) {
        IStableModule stableModule =
 ↪  IStableModule(vault_.moduleAddress(FlatcoinModuleKeys._STABLE_MODULE_KEY));

        InvariantOrder memory invariantBefore = InvariantOrder({
            collateralNet: _getCollateralNet(vault_),
            stableCollateralPerShare: stableModule.stableCollateralPerShare()
        });

        _;

        InvariantOrder memory invariantAfter = InvariantOrder({
            collateralNet: _getCollateralNet(vault_),
            stableCollateralPerShare: stableModule.stableCollateralPerShare()
        });

59:     _collateralNetBalanceRemainsUnchanged(invariantBefore.collateralNet,
 ↪  invariantAfter.collateralNet);
        _stableCollateralPerShareIncreasesOrRemainsUnchanged(
            stableModule.totalSupply(),
            invariantBefore.stableCollateralPerShare,
            invariantAfter.stableCollateralPerShare
        );
        _globalAveragePriceIsNotNegative(vault_);
    }
    function _collateralNetBalanceRemainsUnchanged(int256 netBefore_, int256
 ↪  netAfter_) private pure {
        // Note: +1e6 to account for rounding errors.
        // This means we are ok with a small margin of error such that netAfter -
 ↪  1e6 <= netBefore <= netAfter.
126:    if (netBefore_ > netAfter_ || netAfter_ > netBefore_ + 1e6)
            revert InvariantChecks.InvariantViolation("collateralNet2");
    }
    function _getCollateralNet(IFlatcoinVault vault_) private view returns (int256
 ↪  netCollateral_) {
        int256 collateralBalance =
 ↪  int256(vault_.collateral().balanceOf(address(vault_)));
        int256 trackedCollateral = int256(vault_.stableCollateralTotal()) +
            vault_.getGlobalPositions().marginDepositedTotal;

112:    if (collateralBalance < trackedCollateral) revert
 ↪  InvariantChecks.InvariantViolation("collateralNet1");

        return collateralBalance - trackedCollateral;
    }
```

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

N/A

Impact

Traders could increase her position size as much as she wants without increasing the averageprice. This will ensure the traders get extra margin after settlement of the position and break the invarinat.

In Readme:

> Q: What properties/invariants do you want to hold even if breaking them has a low/unknown impact? Any set of transactions which can break the invariants/properties as mentioned in the InvariantChecks.sol contract file without reverting.

PoC

Due to this issue, trader's EntryPrice could be decreased little by little several times. Then, `Global_pnl` could be less than sum of `pnls`. If all traders close their position, `Global_pnl` could be underwater and deleted with Global_averagePrice. Therefore, stableCollateralPerShare could be decreased. As a result last trader could not close his/her position.

Let's consider this senario: Alice's long position: size = x (>=1e19 = 10 ETH), averagePrice = 1000e18 = $1000. current ETH price = 1000e18 + 1e10. Alice increase her position size by 1e8. newEntryPriceTimesTen = ((averagePrice * size) + (adjustPrice * sizeAdjustment)) * 10 / (size + sizeAdjustment) = = (1000e18 * x + (1000e18 + 1e10) * 1e8) * 10 / (x + 1e8) = = (1000e18 * (x + 1e8) + 1e18) * 10 / (x + 1e8) = 10000e18 + 1e19/(x + 1e8) < 10000e18 + 1 Thus, newEntryPriceTimesTen = 10000e18 and newEntryPrice = 1000e18 = averagePrice. As a result, Alice can increase her position size as much as she wants without increasing the averageprice.

Assuming: stableCollateralTotal = 20 ETH, averagePrice_Global = 1000e18 + 1e10. Alice long position : 10 ETH, 1000$, Bob's long position : 10 ETH, 1000e18 + 2e10 ETH Price : 1000e18 + 1e10

Alice continues to increase her position by 1e8 `x` times at a time. Alice's long position : `size = 10e18 + xe8`, `averagePrice = 1000e18`, Bob's long position: `size = 10e18`, `averagePrice = 1000e18+2e10`. Global: `size = 20e18 + xe8`, `averagePrice = 1000e18 + 1e10`. At this point: Global_pnl and sum of individual_pnl are moving futher and further apart. When last provider close his/her position, individual_pnl is liquidited and Global_pnl is reseted. At this time, the share could be decreased more than 2.

Mitigation

```
LeverageModule.sol
    function executeAdjust(DelayedOrderStructs.Order calldata order_) external
 ↪  onlyAuthorizedModule {
        ...
            uint256 newEntryPriceTimesTen = ((position.averagePrice *
                position.additionalSize +
                adjustPrice *
                uint256(announcedAdjust.additionalSizeAdjustment)) * 10) /
 ↪  newAdditionalSize;

            // In case there is a rounding error, we round up the entry price as
 ↪  this will ensure the traders don't get extra
            // margin after settlement of the position.
-229        newEntryPrice = (newEntryPriceTimesTen % 10 != 0)
-                ? newEntryPriceTimesTen / 10 + 1
-                : newEntryPriceTimesTen / 10;
+           uint256 newEntryAmount = position.averagePrice * position.additionalSize
+               + adjustPrice * uint256(announcedAdjust.additionalSizeAdjustment);
+           newEntryPrice = (newEntryAmount % newAdditionalSize != 0)
+                ? newEntryAmount / newAdditionalSize + 1
+                : newEntryAmount / newAdditionalSize;
```

Test Code

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/tree/main/flatcoin-v1/test/unit/Delayed-Order/DelayedOrder.t.sol#L500 Changed this function to above file.

```
function test_revert_announce_orders_when_global_margin_negative() public {
    uint256 collateralPrice = 1000e8;
    uint256 stableDeposit = 10e18;

    setCollateralPrice(collateralPrice);
    vm.startPrank(admin);

    controllerModProxy.setMaxFundingVelocity(0.03e18 - 1);

    announceAndExecuteDeposit({
        traderAccount: alice,
        keeperAccount: keeper,
        depositAmount: stableDeposit,
        oraclePrice: collateralPrice,
        keeperFeeAmount: 0
    });

    address[10] memory adrs;
    uint256 n = 4;
    uint256 size = 1e18 + 1;

    adrs[1] = makeAddr("trader1");
    adrs[2] = makeAddr("trader2");
    adrs[3] = makeAddr("trader3");
    adrs[4] = makeAddr("trader4");


    for (uint256 i = 1; i <= n; i++) {
        vm.startPrank(admin);
        collateralAsset.transfer(adrs[i], 100e18);
    }
    for (uint256 i = 1; i <= n; i++) {
        announceOpenLeverage(adrs[i], size, size, 0);
    }
    skip(10); // must reach minimum executability time

    uint256[9] memory tokenIds;
    for (uint256 i = 1; i <= n; i++) {
        tokenIds[i] = executeOpenLeverage(keeper, adrs[i], collateralPrice);
    }

    collateralPrice += 1e8 + 1;
    setCollateralPrice(collateralPrice);

    for (uint256 i = 1; i <= n; i++) {
        LeverageModuleStructs.Position memory position =
↪   vaultProxy.getPosition(tokenIds[i]);
        uint256 marginAdjustment = position.marginDeposited / 9;
        uint256 additionalSizeAdjustment = position.additionalSize / 9;
        announceAdjustLeverage(adrs[i], tokenIds[i], int256(marginAdjustment),
↪   int(additionalSizeAdjustment), 0);
    }

    skip(10); // must reach minimum executability time

    for (uint256 i = 1; i <= n; i++) {
        executeAdjustLeverage(keeper, adrs[i], collateralPrice);
    }

    for (uint256 i = n; i >= 1 ; i--) {
        announceCloseLeverage(adrs[i], tokenIds[i], 0);
    }
    skip(10); // must reach minimum executability time

    for (uint256 i = n; i >= 1; i--) {
        executeCloseLeverage(keeper, adrs[i], collateralPrice);
    }
}
```

And change the following two functions in `InvariantChecks.sol`. Because the invariant of `collateralNetBalance` check is first met.

```
LeverageModule.sol
    function executeAdjust(DelayedOrderStructs.Order calldata order_) external
↪   onlyAuthorizedModule {
        ...
            uint256 newEntryPriceTimesTen = ((position.averagePrice *
                position.additionalSize +
                adjustPrice *
                uint256(announcedAdjust.additionalSizeAdjustment)) * 10) /
↪   newAdditionalSize;

            // In case there is a rounding error, we round up the entry price as
↪   this will ensure the traders don't get extra
            // margin after settlement of the position.
-229        newEntryPrice = (newEntryPriceTimesTen % 10 != 0)
-               ? newEntryPriceTimesTen / 10 + 1
-               : newEntryPriceTimesTen / 10;
+           uint256 newEntryAmount = position.averagePrice * position.additionalSize
+                 + adjustPrice * uint256(announcedAdjust.additionalSizeAdjustment);
+             newEntryPrice = (newEntryAmount % newAdditionalSize != 0)
+                 ? newEntryAmount / newAdditionalSize + 1
+                 : newEntryAmount / newAdditionalSize;
```

forge test DelayedOrder.t.sol

Result: [FAIL: InvariantViolation("stableCollateralPerShare")] test_revert_announce_orders_when_global_margin_negative() (gas: 45807281) Logs: averagepriceBob : 1000000020000000000 averagepriceGlobal: 1000000020000000004

You can check that there are no FAIL by using the above mitigated `LeverageModule.sol::executeAdjust()` function.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/dhedge/flatcoin-v1/pull/502

Issue M-17: Bypass The Skew Max.

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/110

The protocol has acknowledged this issue.

Found by

KupiaSec

Summary

The current implementation allows traders to close their positions without checking the `skew`, which can lead to a scenario where the `skew` exceeds the defined limit. As a result, all the functions which include the `skew` check could not run. This causes all functions that rely on `skew` checking to revert.

Root Cause

The issue arises in the `executeClose()` function, where there is no validation of the `skew` before closing a position. This can result in the `skew` exceeding the maximum allowed value.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/blob/main/flatcoin-v1/src/LeverageModule.sol#L335

```
InvariantChecks.sol
    function _collateralNetBalanceRemainsUnchanged(int256 netBefore_, int256
↪   netAfter_) private pure {
        // Note: +1e6 to account for rounding errors.
        // This means we are ok with a small margin of error such that netAfter -
↪   1e6 <= netBefore <= netAfter.
        //if (netBefore_ > netAfter_ || netAfter_ > netBefore_ + 1e6)
        //    revert InvariantChecks.InvariantViolation("collateralNet2");
    }
    function _getCollateralNet(IFlatcoinVault vault_) private view returns (int256
↪   netCollateral_) {
        int256 collateralBalance =
↪   int256(vault_.collateral().balanceOf(address(vault_)));
        int256 trackedCollateral = int256(vault_.stableCollateralTotal()) +
            vault_.getGlobalPositions().marginDepositedTotal;

        //if (collateralBalance < trackedCollateral) revert
↪   InvariantChecks.InvariantViolation("collateralNet1");
```

```
        return collateralBalance - trackedCollateral;
    }
```

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/tree/main/flatcoin-v1/src/abstracts/ControllerBase.sol#L144

```
LeverageModule.sol
299:function executeClose(
        DelayedOrderStructs.Order calldata order_
    ) external onlyAuthorizedModule returns (uint256 marginAfterPositionClose_) {
        IOracleModule oracleModule =
↪   IOracleModule(vault.moduleAddress(FlatcoinModuleKeys._ORACLE_MODULE_KEY));
        DelayedOrderStructs.AnnouncedLeverageClose memory announcedClose =
↪   abi.decode(
            order_.orderData,
            (DelayedOrderStructs.AnnouncedLeverageClose)
        );

        LeverageModuleStructs.Position memory position =
↪   vault.getPosition(announcedClose.tokenId);

        // Make sure the oracle price is after the order executability time
        uint32 maxAge = _getMaxAge(order_.executableAtTime);

        // check that sell price doesn't exceed requested price
        (uint256 exitPrice, ) = oracleModule.getPrice({
            asset: address(vault.collateral()),
            maxAge: maxAge,
            priceDiffCheck: true
        });
        if (exitPrice < announcedClose.minFillPrice)
            revert ICommonErrors.HighSlippage(exitPrice,
↪   announcedClose.minFillPrice);

        uint256 totalFee;
        int256 settledMargin;
        LeverageModuleStructs.PositionSummary memory positionSummary;
        {
            positionSummary = getPositionSummary(position, exitPrice);

            settledMargin = positionSummary.marginAfterSettlement;
            totalFee = announcedClose.tradeFee + order_.keeperFee;

            if (settledMargin <= 0) revert
↪   ICommonErrors.ValueNotPositive("settledMargin");
            // Make sure there is enough margin in the position to pay the keeper
↪   fee
            if (settledMargin < int256(totalFee)) revert
↪   ICommonErrors.NotEnoughMarginForFees(settledMargin, totalFee);
335:        vault.updateStableCollateralTotal(-positionSummary.profitLoss); // pay
↪   the trade fee to stable LPs

            vault.updateGlobalPositionData({
                price: position.averagePrice,
                marginDelta: -(int256(position.marginDeposited) +
↪   positionSummary.accruedFunding),
                additionalSizeDelta: -int256(position.additionalSize)
            });

            // Delete position storage
            vault.deletePosition(announcedClose.tokenId);
        }

        // Cancel any existing limit order on the position
        IOrderAnnouncementModule(vault.moduleAddress(FlatcoinModuleKeys._ORDER_ANNO⌋
↪   UNCEMENT_MODULE_KEY))
            .cancelExistingLimitOrder(announcedClose.tokenId);

        burn(announcedClose.tokenId);

        emit LeverageClose(
            announcedClose.tokenId,
            exitPrice,
            positionSummary,
            uint256(settledMargin),
            position.additionalSize
        );

        return uint256(settledMargin);
    }
OrderExecutionModule.sol
    function _executeLeverageClose(address account_, DelayedOrderStructs.Order
↪   memory order_) internal {
        DelayedOrderStructs.AnnouncedLeverageClose memory leverageClose =
↪   abi.decode(
            order_.orderData,
            (DelayedOrderStructs.AnnouncedLeverageClose)
        );

        ILeverageModule leverageModule =
↪   ILeverageModule(vault.moduleAddress(FlatcoinModuleKeys._LEVERAGE_MODULE_KEY));

        uint256 protocolFeePortion =
↪   FeeManager(address(vault)).getProtocolFee(leverageClose.tradeFee);
        uint256 adjustedTradeFee = leverageClose.tradeFee - protocolFeePortion;

        // Check that position exists (ownerOf reverts if owner is null address)
        // There is a possibility that position was deleted by liquidation or limit
↪   order module
        leverageModule.ownerOf(leverageClose.tokenId);
        uint256 marginAfterPositionClose = leverageModule.executeClose(order_);

        // Collateral and fees settlement.
        {
            // Note: Update the stable collateral total only after trade execution
↪   by the leverage module
            // to avoid any accounting issues.
            vault.updateStableCollateralTotal(int256(adjustedTradeFee));

            // Fees are paid from the remaining position margin.
            vault.sendCollateral({to:
↪   FeeManager(address(vault)).protocolFeeRecipient(), amount: protocolFeePortion});
            vault.sendCollateral({to: msg.sender, amount: order_.keeperFee});

            // Transfer the settled margin minus fee from the vault to the trader.
            vault.sendCollateral({
                to: account_,
                amount: marginAfterPositionClose - leverageClose.tradeFee -
↪   order_.keeperFee
            });
        }
    }
ConrollerBase.sol
    function getProportionalSkew() public view virtual returns (int256 pSkew_) {
        uint256 sizeOpenedTotal = vault.getGlobalPositions().sizeOpenedTotal;
        uint256 stableCollateralTotal = vault.stableCollateralTotal();

        if (stableCollateralTotal > 0) {
144:        pSkew_ = int256(sizeOpenedTotal._divideDecimal(stableCollateralTotal))
↪   - int256(targetSizeCollateralRatio);

            if (pSkew_ < -1e18 || pSkew_ > 1e18) {
                pSkew_ = DecimalMath.UNIT.min(pSkew_.max(-DecimalMath.UNIT));
            }
        } else {
            assert(sizeOpenedTotal == 0);
            pSkew_ = 0;
        }
    }
```

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

N/A

Impact

Procotol cannot function as intended. The market may close if the price of ETH exceeds six times its opening price.

PoC

Let's consider this senario. Assuming: `maxskewFraction = 120%`, `stableCollateralTotal := 100ETH`, `_globalPositions.sizeOpenedTotal := 120ETH`, `currentPrice := 4400$`. Alice's old long position: `size = 110 ETH`, `averagePrice = $600`, Bob's long position: `size = 10 ETH`, `averagePrice = $4000` Then, Alice's `pnl = 110 * (4400 - 600) / 4400 = 95 ETH`. After Alice closes her position: `stableCollateralTotal = 5 ETH`, `sizeOpenedTotal = 10 ETH`. At this time, `skew = 10 / 5 = 2 > maxskewFraction`.

Another senario. Assuming: `stableCollateralTotal := 100ETH`, `_globalPositions.sizeOpenedTotal := 120ETH`, `currentPrice := 4400$`, Alice's old long position: `size = 110 ETH`, `averagePrice = $400`. Bob's long position: `size = 10 ETH`, `averagePrice = $4000` Then, Alice's `pnl = 110 * (4400 - 400) / 4400 = 100 ETH`. After Alice closes her position: `stableCollateralTotal = 0`, `sizeOpenedTotal = 10 ETH`. After this time, skew calcuation is reverted due to the getProportionalSkew() function. And this function is used for `Borrow Rate Fee` calculation in the `settleFundingFees()` function. This means that protol is broken.

Duplication Clarification

In sherlock docs:
https://docs.sherlock.xyz/audits/judging/guidelines#ix.-duplication-guidelines

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest-judging/issues/12

- Root Cause Identification My Report: Identifies that the `skew` is increased due to the `executeLeverageClose()` function, which allows positions to close without `skew` checks. Linked Report: Focuses on incorrect `skew` calculations by design, which is a separate issue.
- Non-Duplication Justification Since My report addresses the lack of preventive measures in closing positions while the linked report discusses calculation

inaccuracies, they tackle different aspects of the `skew` issue and should not be considered duplicates.

Mitigation

1. Consider the using of actual liquidit for Skew calculations instead of `stableCollateralTotal`,
2. or add more `skew` checks in multiple places.

https://github.com/sherlock-audit/2023-12-flatmoney-judging/issues/186 https://github.com/sherlock-audit/2023-12-flatmoney-judging/issues/161 The root cause of these issues lies in misapplication, rather than the application itself.

Issue M-18: `Borrow Rate Fee` Is Incorrect.

Source:
https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update-judging/issues/113

The protocol has acknowledged this issue.

Found by

KupiaSec

Summary

The current implementation incorrectly affects new users due to the accumulated pnl of previous users.

For new users, the accumulated `pnl` is not relevant. However, in the current implementation, it does affect it. Because the `stableCollateralTotal` is `totalAfterSettlement` plus `total_pnl`. As a result, `Borrow Rate Fee` incorrectly applied.

Root Cause

The issue stems from how the `stableCollateralTotal` is calculated and used. The accumulated `pnl` should not influence new users, but it currently does.

https://github.com/sherlock-audit/2024-12-flat-money-v1-1-update/tree/main/flatcoin-v1/src/abstracts/ControllerBase.sol#L144

```
ControllerBase.sol
207:function profitLossTotal(uint256 price_) public view virtual returns (int256
↪   pnl_) {
        FlatcoinVaultStructs.GlobalPositions memory globalPosition =
↪   vault.getGlobalPositions();
        int256 priceShift = int256(price_) - int256(globalPosition.averagePrice);

        return (int256(globalPosition.sizeOpenedTotal) * (priceShift)) /
↪   int256(price_);
    }
    function getProportionalSkew() public view virtual returns (int256 pSkew_) {
        uint256 sizeOpenedTotal = vault.getGlobalPositions().sizeOpenedTotal;
        uint256 stableCollateralTotal = vault.stableCollateralTotal();

        if (stableCollateralTotal > 0) {
144:        pSkew_ = int256(sizeOpenedTotal._divideDecimal(stableCollateralTotal))
↪   - int256(targetSizeCollateralRatio);

            if (pSkew_ < -1e18 || pSkew_ > 1e18) {
                pSkew_ = DecimalMath.UNIT.min(pSkew_.max(-DecimalMath.UNIT));
            }
        } else {
            assert(sizeOpenedTotal == 0);
            pSkew_ = 0;
        }
    }

StableModule.sol
181:function stableCollateralTotalAfterSettlement(
        uint32 maxAge_,
        bool priceDiffCheck_
    ) public view returns (uint256 stableCollateralBalance_) {
        // Assumption => pnlTotal = pnlLong + fundingAccruedLong
        // The assumption is based on the fact that stable LPs are the counterparty
 ↪  to leverage traders.
        // If the `pnlLong` is +ve that means the traders won and the LPs lost
 ↪  between the last funding rate update and now.
        // Similary if the `fundingAccruedLong` is +ve that means the market was
 ↪  skewed short-side.
        // When we combine these two terms, we get the total profit/loss of the
 ↪  leverage traders.
        // NOTE: This function if called after settlement returns only the PnL as
 ↪  funding has already been adjusted
        //      due to calling `_settleFundingFees()`. Although this still means
 ↪  `netTotal` includes the funding
        //      adjusted long PnL, it might not be clear to the reader of the code.
        int256 netTotal = IControllerModule(vault.moduleAddress(FlatcoinModuleKeys.┘
 ↪  _CONTROLLER_MODULE_KEY))
            .fundingAdjustedLongPnLTotal({maxAge: maxAge_, priceDiffCheck:
 ↪  priceDiffCheck_});

        // The flatcoin LPs are the counterparty to the leverage traders.
        // So when the traders win, the flatcoin LPs lose and vice versa.
        // Therefore we subtract the leverage trader profits and add the losses
        int256 totalAfterSettlement = int256(vault.stableCollateralTotal()) -
 ↪  netTotal;

        if (totalAfterSettlement < 0) {
            stableCollateralBalance_ = 0;
        } else {
            stableCollateralBalance_ = uint256(totalAfterSettlement);
        }
    }
```

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

N/A

Impact

Incorrect using of `Borrow Rate Fee` always cause loss for one side, either traders or liquidity providers

PoC

Scenario1: Assuming: `stableCollateralTotal = 105 ETH`, `_globalPositions.sizeOpenedTotal = 100 ETH`, `_globalPositions.averagePrice = $4500`, `currentPrice = $5000`, `targetSizeCollateralRatio = 1e18`, `total shares = 4.5e5 (UNIT)`, `Alice's shares = 4.5e5`. `total pnl = 100 * (5000 - 4500) / 5000 = 10 ETH`, `totalAfterSettlement = 105 - 10 = 95 ETH`.

Bob deposits 95 ETH: `Bob's shares = 95 / (95 / 4.5e5) = 4.5e5`. `stableCollateralTotal = 200 ETH`, `total pnl = 10 ETH`, `totalAfterSettlement = 190 ETH`, `skew = 100/200 - 1 < 0`. The `stableCollateralTotal` includes Alice's 105 ETH, and Bob's 95 ETH(however, the ETH prices are different). But Alice and Bob are paying same 'Borrow Rate Fee'(Because their shares are same). This is a loss for Bob.

Alice withdraw her 4.5e5(UNIT) shares. `stableCollateralTotal = 105 ETH`, `total pnl = 10 ETH`, `totalAfterSettlement = 95 ETH`, `skew = 100/105 - 1 < 0`. Bob deposited only 95 ETH(less than sizeOpenTotal), but now he is paying the `Borrow Rate Fee` instead of receiving it. This is another loss for Bob.

Scenario2: Assuming: `stableCollateralTotal = 110 ETH`, `_globalPositions.sizeOpenedTotal = 100 ETH`, `_globalPositions.averagePrice = $4600`, `currentPrice = $4000`, `targetSizeCollateralRatio = 1e18`, `total shares = 4.6e5 (UNIT)`, `Alice's shares = 4.6e5`. `total pnl = 100 * (4000 - 4600) / 4000 = -15 ETH`, `totalAfterSettlement = 110 - (-15) = 125 ETH`.

Bob deposits 110 ETH: `Bob's shares = 110 / (125 / 4.6e5) = 4.048e5`. `stableCollateralTotal = 220 ETH`, `total pnl = -15 ETH`, `totalAfterSettlement = 235 ETH`, `skew = 100/235 - 1 < 0`. The `stableCollateralTotal` includes Alice's 110 ETH, and Bob's 110 ETH(however, the ETH prices are different). But Alice is paying more 'Borrow Rate Fee' than Bob(because, Alice's shares > Bob's shares). This is a loss for Alice.

Alice withdraw her 4.6e5(UNIT) shares. `stableCollateralTotal = 95 ETH`, `total pnl = -15 ETH`, `totalAfterSettlement = 110 ETH`, `skew = 100/95 - 1 > 0`. Bob deposited 110 ETH(more than sizeOpenTotal), but now he is receiving the `Borrow Rate Fee` instead of paying. This is a loss for traders.

This problem exists even when viewed from the trader's perspective.

This means that the relationships between old providers and old traders are influencing new users. For new users, a new market or an old market should be the same, but that is not the case. This is because of Alice, and this situation should reset when Alice leaves, but the current implementation does not allow for that.

Scenario3: Assuming: `stableCollateralTotal := 85 ETH`, `currentPrice := $4000`, `maxskewFraction = 120%`. Trader Alice's old long position: `size := 100 ETH`, `averagePrice := $600` Trader Bob's old long position: `size := 2 ETH`, `averagePrice := $2000` `Alice's pnl = 100 * (4000 - 600) / 4000 = 85 ETH`. `Bob's pnl = 2 * (4000 - 2000) / 4000 = 1 ETH`. If Alice closes his position, then: `stableCollateralTotal = 0`, `totalAfterSettlement = -1(ETH) -> 0`. At this time `Borrow Rate Fee` can't calculate. Bob can't close his position and because `stableCollateralPerShare = 0` anyone can't mint shares. As a result, the protocol is broken.

Duplication Clarification

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest-judging/issues/11 My Report: Focuses on how accumulated pnl impacts new users in the market. Linked Report: Identifies parties responsible for causing the pnl. These reports address different aspects; hence, they should not be considered duplicates.

Mitigation

1. Consider settling the `total_pnl` periodically,
2. or using providers' real funds instead of `stableCollateralTotal`.

https://github.com/sherlock-audit/2023-12-flatmoney-judging/issues/186 https://github.com/sherlock-audit/2023-12-flatmoney-judging/issues/161 The root cause of these issues lies in misapplication, rather than the application itself.

Let's consider settling the `total_pnl` exactly periodically. 1.

```
struct GlobalPositions {
    int256 marginDepositedTotal;
+   uint touchedPnl;
    uint256 averagePrice;
    uint256 sizeOpenedTotal;
}
```

2. stableCollateralTotal = stableCollateralTotal + touchedPnl - total_pnl.
3. marginDepositedTotal = marginDepositedTotal - touchedpnl + total_pnl.
4. touchedPnl = total_pnl.

This will be mitigated many issues.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.