SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: GMX
Prepared by: Sherlock
Lead Security Expert: lllllll
Dates Audited: April 25 - June 4, 2023
Prepared on: July 19, 2023

Introduction

GMX is a decentralized spot and perpetual exchange that supports low swap fees and zero price impact trades.

Scope

Repository: gmx-io/gmx-synthetics

Branch: fix-review

Commit: a2e331f6d0a3b59aaac5ead975b206840369a723

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
11	5

Security experts who found valid issues

rvierdiiev
J4de
Ch_301
ShadowForce
chaduke
Nyx
0xGoodess
KingNFT
Chinmay
bin2chen
0xdeadbeef
lemonmon
lllllll
stent
ten-on-ten

Issue H-1: MarketUtils.getPoolValueInfo() does not use !maximize when evaluating impactPoolUsd, leading to wrong logic of maximizing or minimizing the pool value.

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/160

Found by

chaduke

Summary

`MarketUtils.getPoolValueInfo()` does not use `!maximize` when evaluating `impactPoolUsd`, leading to wrong logic of maximizing or minimizing the pool value.

This function is reachable by a withdrawal order to determine the amount of long/short tokens to withdraw, as a result, such amounts might be overesitmated due to the wrong logic in `MarketUtils.getPoolValueInfo()`, leading to possible draining of the pool in the long run!

Vulnerability Detail

`MarketUtils.getPoolValueInfo()` is used to get the USD value of a pool. It is called in a withdrawal order execution to determine the amount of long/short tokens to withdraw via flow `WithdrawalUtils.executeWithdrawal() -> _executeWithdrawal() -> WithdrawUtils.executeWithdrawal() -> _executeWithdrawal() -> _getOutputAmounts() -> MarketUtils.getPoolValueInfo()`. Let's look at `MarketUtils.getPoolValueInfo()` assuming maximize = false. (which is the case in a withdrawl order). That is, we are trying to minimize the pool value. This is understandable since we need to minimize the output amounts of withdrawn tokens in favor of the system. The analysis for the case of maximize = true is similar.

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L289-L375

There are a few items we need to calculate:

1. longTokenUsd: we pick price longTokenPrice.pickPrice(maximize);
2. shortTokenUsd: we pick price shortTokenPrice.pickPrice(maximize)
3. totalBorrowingFee for the long side, no optimization option.
4. totalBorrowingFee for the short side, no optimization option.
5. impactPoolUsd, indexTokenPrice.pickPrice(maximize) is chosen, which is wrong, since this is to be subtracted from the total Usd value, we need to use

indexTokenPrice.pickPrice(!maximize), just like the next item for calculating Pnl.

6. longPnl and shortPnl, both use the mode `!maximize` since both of them will be subtracted from the total value.

In summary, just like calculating Pnl, we need to use `indexTokenPrice.pickPrice(!maximize)` instead of `indexTokenPrice.pickPrice(maximize)` to calculate `impactPoolUsd`. Only in this way, the logic of the input parameter `maximize` can be implemented properly.

Impact

Code Snippet

getPoolValueInfo() does not use !maximize when evaluating impactPoolUsd, leading to wrong logic of maximizing or minimizing the pool value. As a result, when `isMaximize` is true, the returned value is actually not maximized! In the case of withdrawl, a slight overestimation of the output tokens might occur and lead to possible draining of the pool in the long run!

Tool used

VSCode

Manual Review

Recommendation

In function `MarketUtils.getPoolValueInfo()` we need to use `indexTokenPrice.pickPrice(!maximize)` instead of `indexTokenPrice.pickPrice(maximize)` to calculate `impactPoolUsd`.

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/commit/13913a28e4b07f5a2cc0065fdebc34c437864c71

Issue H-2: Pool amount adjustments for collateral decreases aren't undone if swaps are successful

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/235

Found by

lllllll, bin2chen, stent

Summary

Pool amount adjustments, which are added to storage as a temporary accounting tracking mechanism, aren't undone if swaps are successful

Vulnerability Detail

`swapProfitToCollateralToken()` uses `Keys.poolAmountAdjustmentKey()` to temporarily store an adjustment to the pool amount, and undoes the adjustment at the end of the function, after the try-catch block. However, the function bypasses the end of the function in the succeess case, because it returns early, and therefore doesn't undo the adjustment.

Impact

The value is looked up and used in `getNextPoolAmountsParams()` for swap orders that occur afterwards. The adjusted amount will be included until someone does another swap of the profit to a collateral token, at which point it will have the new value (and not be reset unless there is an exception). The adjustment ends up being used in the calculation of swap impact, so all subsequent swaps will be priced incorrectly, giving some users discounts they don't deserve, and others a penalty that they don't deserve, when doing swaps.

Code Snippet

Adjustment is added to storage, but isn't undone if `swapHandler.swap()` is successful:

```
// File: gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol :
↪  DecreasePositionCollateralUtils.swapProfitToCollateralToken()   #1

452                 // adjust the pool amount by the poolAmountDelta so that the
↪  price impact of the swap will be
453                 // more accurately calculated
454 @>              params.contracts.dataStore.setInt(Keys.poolAmountAdjustmentKe
↪  y(params.market.marketToken, pnlToken), poolAmountDelta);
```

```
455
456                 try params.contracts.swapHandler.swap(
457                     SwapUtils.SwapParams(
...
470                     )
471                 ) returns (address /* tokenOut */, uint256 swapOutputAmount) {
472 @>                  return (true, swapOutputAmount);
473                 } catch Error(string memory reason) {
474                     emit SwapUtils.SwapReverted(reason, "");
475                 } catch (bytes memory reasonBytes) {
476                     (string memory reason, /* bool hasRevertMessage */) =
 ↪  ErrorUtils.getRevertMessage(reasonBytes);
477                     emit SwapUtils.SwapReverted(reason, reasonBytes);
478                 }
479
480 @>              params.contracts.dataStore.setInt(Keys.poolAmountAdjustmentKe
 ↪  y(params.market.marketToken, pnlToken), 0);
481             }
482
483             return (false, 0);
484:        }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol#L452-L484

Tool used

Manual Review

Recommendation

Undo the adjustment before returning

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/pull/155/commits/cd9f9c2f0f025c819f1080332514d6a2dedc05fc

the previous flow of updating a global variable seemed more of a workaround to fix the pool amount for the swap

the code was updated to use a more straightforward solution of updating the pool amount directly before performing the swap

Issue H-3: Swaps associated with position orders will use the wrong price

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/240

Found by

lllllll

Summary

Swaps of tokens gained via position orders will use the wrong price as the latest price

Vulnerability Detail

In the previous iteration of the code, the `getLatestPrice()` function returned the secondary price, and fell back to the primary price if the secondary price didn't exist. In the current version of the code, `getLatestPrice()` returns the custom price if one exists, which may be the trigger price or the maximized price. This price is used not only for the execution of the order, but also now for the swaps of the tokens after the order executes.

Impact

If, for example, the order is a market increase order, the custom price is set to the maximized price for the execution which means the liquidity taker got fewer shares than the maker gave. When those shares are swapped, that maximized price is still used, whereas if the swap had been done as a separate order, no custom price would be consulted. The two methods of doing swaps get different prices, which leads to arbitrage opportunities.

Code Snippet

Normal swap orders never have an exact price set:

```
// File: gmx-synthetics/contracts/order/BaseOrderUtils.sol :
↪  BaseOrderUtils.setExactOrderPrice()   #1

203         function setExactOrderPrice(
204             Oracle oracle,
205             address indexToken,
206             Order.OrderType orderType,
207             uint256 triggerPrice,
208             bool isLong
```

```
209         ) internal {
210             if (isSwapOrder(orderType)) {
211 @>              return;
212:            }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/order/BaseOrderUtils.sol#L203-L212

but swaps after the order will use the price set during the position-altering portion of the order:

```
// File: gmx-synthetics/contracts/swap/SwapUtils.sol : SwapUtils._swap()   #2

178         function _swap(SwapParams memory params, _SwapParams memory _params)
 ↪  internal returns (address, uint256) {
179             SwapCache memory cache;
180
181             if (_params.tokenIn != _params.market.longToken &&
 ↪  _params.tokenIn != _params.market.shortToken) {
182                 revert Errors.InvalidTokenIn(_params.tokenIn,
 ↪  _params.market.marketToken);
183             }
184
185             MarketUtils.validateSwapMarket(_params.market);
186
187             cache.tokenOut = MarketUtils.getOppositeToken(_params.tokenIn,
 ↪  _params.market);
188 @>          cache.tokenInPrice =
 ↪  params.oracle.getLatestPrice(_params.tokenIn);
189:@>          cache.tokenOutPrice =
 ↪  params.oracle.getLatestPrice(cache.tokenOut);
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/swap/SwapUtils.sol#L178-L189

Tool used

Manual Review

Recommendation

Introduce a flag to the `getLatestPrice()` function, indicating whether to use the custom price if it exists

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/commit/3243138ebdc6f86e06f5b1fef91312113ef36e20

the previous logic of allowing multiple token types led to logical issues and unexpected pricing differences

the code was updated to use a single (min, max) price value per token instead for each transaction

Issue H-4: Limit swap orders can be used to get a free look into the future

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/241

Found by

0xdeadbeef, lllllll, Nyx

Summary

Users can cancel their limit swap orders to get a free look into prices in future blocks

Vulnerability Detail

This is a part of the same issue that was described in the last contest. The sponsor fixed the bug for `LimitDecrease` and `StopLossDecrease`, but not for `LimitSwap`.

Any swap limit order submitted in block range N can't be executed until block range N+2, because the block range is forced to be after the submitted block range, and keepers can't execute until the price has been archived, which necessarily won't be until *after* block range N+1. Consider what happens when half of the oracle's block ranges are off from the other half, e.g.:

```
    1 2 3 4 5 6 7 8 9 < block number
O1: A B B B B C C C D
O2: A A B B B B C C C
^^ grouped oracle block ranges
```

At block 1, oracles in both groups (O1 and O2) are in the same block range A, and someone submits a large swap limit order (N). At block 6, oracles in O1 are in N+2, but oracles in O2 are still in N+1. This means that the swap limit order will execute at the median price of block 5 (since the earliest group to have archive prices at block 6 for N+1 will be O1) and market swap order submitted at block 6 in the other direction will execute at the median price of block 6 since O2 will be the first group to archive a price range that will contain block 6. By the end of block 5, the price for O1 is known, and the price that O2 will get at block 6 can be predicted with high probability (e.g. if the price has just gapped a few cents), so a trader will know whether the two orders will create a profit or not. If a profit is expected, they'll submit the market order at block 6. If a loss is expected, they'll cancel the swap limit order from block 1, and only have to cover gas fees.

Essentially the logic is that limit swap orders will use earlier prices, and market orders (with swaps) will use later prices, and since oracle block ranges aren't fixed,

an attacker is able to know both prices before having their orders executed, and use large order sizes to capitalize on small price differences.

Impact

There is a lot of work involved in calculating statistics about block ranges for oracles and their processing time/queues, and ensuring one gets the prices essentially when the keepers do, but this is likely less work than co-located high frequency traders in traditional finance have to do, and if there's a risk free profit to be made, they'll put in the work to do it every single time, at the expense of all other traders.

Code Snippet

Market orders can use the current block, but limit orders must use the next block:

```
// File: gmx-synthetics/contracts/order/SwapOrderUtils.sol :
↪  SwapOrderUtils.validateOracleBlockNumbers()   #1

57            if (orderType == Order.OrderType.MarketSwap) {
58 @>             OracleUtils.validateBlockNumberWithinRange(
59                    minOracleBlockNumbers,
60                    maxOracleBlockNumbers,
61                    orderUpdatedAtBlock
62                );
63                return;
64            }
65
66            if (orderType == Order.OrderType.LimitSwap) {
67 @>             if
↪  (!minOracleBlockNumbers.areGreaterThan(orderUpdatedAtBlock)) {
68                    revert
↪  Errors.OracleBlockNumbersAreSmallerThanRequired(minOracleBlockNumbers,
↪  orderUpdatedAtBlock);
69                }
70                return;
71            }
72
73            revert Errors.UnsupportedOrderType();
74:       }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/order/SwapOrderUtils.sol#L57-L74

Tool used

Manual Review

Recommendation

All orders should follow the same block range rules

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/commit/c5fdc2952a8c982a7c215477124a627699c2009c

IIIIIII000

https://github.com/gmx-io/gmx-synthetics/commit/c5fdc2952a8c982a7c215477124a627699c2009c The commit changes from a greater-than block number check to a greater-than-or-equal-to check. `Array.areGreaterThan()` is no longer called anywhere, so this specific issue is resolved. done

Issue H-5: `initialCollateralDeltaAmount` is incorrectly interpreted as a USD value when calculating estimated remaining collateral

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/249

Found by

Chinmay, lllllll, bin2chen, ten-on-ten

Summary

Decrease orders have checks to ensure that if collateral is withdrawn, that there is enough left that the liquidation checks will still pass. The code that calculates the remaining collateral incorrectly adds a token amount to a USD value.

Vulnerability Detail

`initialCollateralDeltaAmount` is incorrectly interpreted as a USD value when calculating estimated remaining collateral which means, depending on the token's decimals, the collateral will either be accepted or not accepted, when it shouldn't be.

Impact

If the remaining collateral is over-estimated, the `MIN_COLLATERAL_USD` checks later in the function will pass, and the user will be able decrease their collateral, but then will immediately be liquidatable by liquidation keepers, since liquidation orders don't attempt to change the collateral amount.

If the remaining collateral is under-estimated, the user will be incorrectly locked into their position.

Code Snippet

`initialCollateralDeltaAmount()` isn't converted to a USD amount before being added to `estimatedRemainingCollateralUsd`, which is a USD amount:

```
// File: gmx-synthetics/contracts/position/DecreasePositionUtils.sol :
↪  DecreasePositionUtils.decreasePosition()   #1

139                   // the estimatedRemainingCollateralUsd subtracts the
↪  initialCollateralDeltaAmount
140                   // since the initialCollateralDeltaAmount will be set to
↪  zero, the initialCollateralDeltaAmount
```

```
141                     // should be added back to the
↪ estimatedRemainingCollateralUsd
142 @>                  estimatedRemainingCollateralUsd +=
↪ params.order.initialCollateralDeltaAmount().toInt256();
143                     params.order.setInitialCollateralDeltaAmount(0);
144                 }
145
146                 // if the remaining collateral including position pnl will be
↪ below
147                 // the min collateral usd value, then close the position
148                 //
149                 // if the position has sufficient remaining collateral
↪ including pnl
150                 // then allow the position to be partially closed and the
↪ updated
151                 // position to remain open
152:@>              if ((estimatedRemainingCollateralUsd +
↪ cache.estimatedRemainingPnlUsd) <
↪ params.contracts.dataStore.getUint(Keys.MIN_COLLATERAL_USD).toInt256()) {
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/position/DecreasePositionUtils.sol#L132-L152

Tool used

Manual Review

Recommendation

Convert to a USD amount before doing the addition

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/commit/f8f2dd694269a8a795a45ed961fb9970e9ce3cc8

IIIIIIIOOO

https://github.com/gmx-io/gmx-synthetics/commit/f8f2dd694269a8a795a45ed961fb9970e9ce3cc8 The commit implements the suggested fix of converting the amount to a USD value, and uses the collateral token's min price as the conversion price. The resulting value is never written back to storage, and is only used as a minimum collateral threshold check, so using this less-favorable-to-the-user price is correct. Looking at 97c826246b06fc977191b8090c970c0ff93cf88a, later changes moved the price lookup to the top of the `decreasePosition()` function, and

changes from using `MarketUtils.getMarketPricesForPosition()` to using `MarketUtils.getMarketPrices()`, which means the primary price is now always used, rather than the latest price which may have been a custom or secondary price. According to the comments and fix for #240, there is no longer a custom or secondary price, so the change for this fix looks correct. done

Issue M-1: User can loose funds in case if swapping in DecreaseOrderUtils.processOrder will fail

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/50

Found by

rvierdiiev

Summary

When user executes decrease order, then he provides `order.minOutputAmount` value, that should protect his from loses. This value is provided with hope that swapping that will take some fees will be executed. But in case if swapping will fail, then this `order.minOutputAmount` value will be smaller then user would like to receive in case when swapping didn't occur. Because of that user can receive less output amount.

Vulnerability Detail

`DecreaseOrderUtils.processOrder` function executed decrease order and returns order execution result which contains information about output tokens and amounts that user should receive.

In case if only 1 output token is returned to user, then function will try to swap that amount according to the swap path that user has provided.
https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/order/DecreaseOrderUtils.sol#L83-L116

```
    try params.contracts.swapHandler.swap(
        SwapUtils.SwapParams(
            params.contracts.dataStore,
            params.contracts.eventEmitter,
            params.contracts.oracle,
            Bank(payable(order.market())),
            params.key,
            result.outputToken,
            result.outputAmount,
            params.swapPathMarkets,
            0,
            order.receiver(),
            order.uiFeeReceiver(),
            order.shouldUnwrapNativeToken()
        )
    ) returns (address tokenOut, uint256 swapOutputAmount) {
        `(
```

```
            params.contracts.oracle,
            tokenOut,
            swapOutputAmount,
            order.minOutputAmount()
        );
    } catch (bytes memory reasonBytes) {
        (string memory reason, /* bool hasRevertMessage */) =
↪   ErrorUtils.getRevertMessage(reasonBytes);

        _handleSwapError(
            params.contracts.oracle,
            order,
            result,
            reason,
            reasonBytes
        );
    }
}
```

As you can see in case if swap succeeded, then `_validateOutputAmount` function will be called, that will check slippage. It will check that `swapOutputAmount` is received according to the slippage. But in case if swap will not succeed, then `_handleSwapError` will be called. https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/order/DecreaseOrderUtils.sol#L208-L230

```
null(
    Oracle oracle,
    Order.Props memory order,
    DecreasePositionUtils.DecreasePositionResult memory result,
    string memory reason,
    bytes memory reasonBytes
) internal {
    emit SwapUtils.SwapReverted(reason, reasonBytes);

    _validateOutputAmount(
        oracle,
        result.outputToken,
        result.outputAmount,
        order.minOutputAmount()
    );

    MarketToken(payable(order.market())).transferOut(
        result.outputToken,
        order.receiver(),
        result.outputAmount,
        order.shouldUnwrapNativeToken()
    );
```

```
}
```

As you can see in this case `_validateOutputAmount` function will be called as well, but it will be called with `result.outputAmount` this time, which is amount provided by decreasing of position.

Now i will describe the problem. In case if user wants to swap his token, he knows that he needs to pay fees to the market pools and that this swap will eat some amount of output. So in case if `result.outputAmount` is 100$ worth of tokenA, it's fine if user will provide slippage as 3% if he has long swap path, so his slippage is 97. $But in case when swap will fail, then now this slippage of 97$ is incorrect as user didn't do swapping and he should receiev exactly 100$ worth of tokenA.

Also i should note here, that it's easy to make swap fail for keeper, it's enough for him to just not provide any asset price, so swap reverts. So keeper can benefit on this slippage issue.

Impact

User can be frontruned to receive less amount in case of swapping error.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Maybe it's needed to have another slippage param, that should be used in case of no swapping.

Discussion

xvi10

would classify this as a low, the prices provided must still be within the max oracle age which could be a few minutes, it should be difficult to intentionally cause failures within this range

IIIIIII000

It still sounds like a medium to me. I'll let Sherlock decide

xvi10

> Also i should note here, that it's easy to make swap fail for keeper, it's enough for him to just not provide any asset price, so swap reverts. So keeper can benefit on this slippage issue.

keepers do not benefit from failed swaps, but okay with me to let Sherlock decide for this one

hrishibhat

Given that there is a loss of funds for the user in the unlikely case of swaps failing, considering this issue a valid medium

rvierdiyev

Escalate for 10 USDC I don't think that #124 is duplicate of this issue.

I think that this report and #124 describe different things. After reading #124 i didn't feel that it's same as this report, because here i describe error case of swapping. While #124 is talking about swapping to different token.

I believe that these 2 should be separate issues.

sherlock-admin

> Escalate for 10 USDC I don't think that #124 is duplicate of this issue.
>
> I think that this report and #124 describe different things. After reading #124 i didn't feel that it's same as this report, because here i describe error case of swapping. While #124 is talking about swapping to different token.
>
> I believe that these 2 should be separate issues.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

IIIIIII000

Agree with the escalation - #124 is Invalid, and this is a solo Medium

hrishibhat

Result: Medium Unique #124 is not a duplicate of this issue

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- rvierdiyev: accepted

Issue M-2: MarketUtils.getFundingAmountPerSizeDelta() has a rounding logical error.

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/164

Found by

chaduke

Summary

`MarketUtils.getFundingAmountPerSizeDelta()` has a rounding logical error. The main problem is the divisor always use a roundupDivision regardless of the input `roundUp` rounding mode. Actually the correct use should be: the divisor should use the opposite of `roundup` to achieve the same logic of rounding.

Vulnerability Detail

`MarketUtils.getFundingAmountPerSizeDelta()` is used to calculate the `FundingAmountPerSizeDelta` with a roundup input mode parameter.

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L1126-L1136

This function is used for example by the IncreaseLimit order via flow `OrderHandler.executeOrder() -> _executeOrder() -> OrderUtils.executeOrder() -> processOrder() -> IncreaseOrderUtils.processOrder() -> IncreasePositionUtils.increasePosition() -> PositionUtils.updateFundingAndBorrowingState() -> MarketUtils.updateFundingAmoutPerSize() -> getFundingAmountPerSizeDelta()`.

However, the main problem is the divisor always use a roundupDivision regardless of the input `roundUp` rounding mode. Actually the correct use should be: the divisor should use the opposite of `roundup` to achieve the same logic of rounding.

My POC code confirms my finding: given fundingAmount = 2e15, openInterest = 1e15+1, and roundup = true, the correct answer should be: 19999999999999980000000000000019999999999999999. However, the implementation returns the wrong solution of : 10000000000000000000000000000000000000000000000. The reason is that the divisor uses a roundup and gets a divisor of 2, as a result, the final result is actually rounded down rather than rounding up!

```
function testGetFundingAmountPerSizeDelta() public{
        uint result = MarketUtils.getFundingAmountPerSizeDelta(2e15, 1e15+1,
↪   true);
        console2.log("result: %d", result);
```

```
        uint256 correctResult = 2e15 * 1e15 * 1e30 + 1e15; // this is a real
↪   round up
        correctResult = correctResult/(1e15+1);
        console2.log("correctResult: %d", correctResult);
        assertTrue(result  == 1e15 * 1e30);
    }
```

Impact

MarketUtils.getFundingAmountPerSizeDelta() has a rounding logical error, sometimes, when roundup = true, the result, instead of rounding up, it becomes a rounding down!

Code Snippet

Tool used

VScode

Manual Review

Recommendation

Change the rounding mode of the divisor to the opposite of the input `roundup` mode. Or, the solution can be just as follows:

```
function getFundingAmountPerSizeDelta(
        uint256 fundingAmount,
        uint256 openInterest,
        bool roundUp
    ) internal pure returns (uint256) {
        if (fundingAmount == 0 || openInterest == 0) { return 0; }



        // how many units in openInterest
-        uint256 divisor = Calc.roundUpDivision(openInterest,
↪   Precision.FLOAT_PRECISION_SQRT);

-        return Precision.toFactor(fundingAmount, divisor, roundUp);
+       return Precision.toFactor(fundingAmount*Precision.FLOAT_PRECISION_SQRT,
↪   openInterest, roundUp
    }
```

Discussion

xvi10

would classify this as a low since the impact should be very small

IIIIIII000

a fix was done for this one, so I don't think this can be low: https://github.com/gmx-io/gmx-synthetics/commit/f8cdb4df7bba9718fe16f3ab78ebd1c9cd0b2bc6

xvi10

we will fix valid low issues if it can improve the accuracy of calculations, even if the issue may not lead to a significant difference

was referencing this for the criteria, "Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future."

the rounding issues need to lead to a material impact to be considered for a medium?

IIIIIII000

won't this lead to funding miscalculations for every second, for every user, and become larger as the funding amounts grow?

xvi10

that's true, have updated the issue to confirmed

IIIIIII000

@xvi10 Also, if the precision loss is enough to make you want to add a fix, I would say it should be medium - let me know if you intend to fix any of the other precision loss submissions

xvi10

not sure i agree with that since it doesn't seem to match the criteria in https://docs.sherlock.xyz/audits/judging/judging

but will notify and let Sherlock decide

the MarketUtils.getFundingFeeAmount was also updated, not due to precision but to avoid the risk of overflow, it was mentioned as a precision issue in: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/193

hrishibhat

Considering this issue as valid medium based on the following comment which the sponsor agrees too.

won't this lead to funding miscalculations for every second, for every user, and become larger as the funding amounts grow?

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/pull/155/commits/f8cdb4df7bba9718fe16f3ab78ebd1c9cd0b2bc6

Issue M-3: PositionUtils.validatePosition() uses `isIncrease` instead of `false` when calling isPositionLiquidatable(), making it not work properly for the case of `isIncrease = true`.

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/180

Found by

chaduke

Summary

`PositionUtils.validatePosition()` uses `isIncrease` instead of `false` when calling isPositionLiquidatable(), making it not work properly for the case of `isIncrease = true`. The main problem is that when calling `isPositionLiquidatable()`, we should always consider decreasing the position since we are proposing a liquidation trade (which is a decrease in position). Therefore, it should not use `isIncrease` for the input parameter for `isPositionLiquidatable()`. We should always use `false` instead.

Vulnerability Detail

`PositionUtils.validatePosition()` is called to validate whether a position is valid in both collateral size and position size, and in addition, to check if the position is liquidable:

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/position/PositionUtils.sol#L261-L296

It calls function `isPositionLiquidatable()` to check if a position is liquidable. However, it passes the `isIncrease` to function `isPositionLiquidatable()` as an argument. Actually, the `false` value should always be used for calling function `isPositionLiquidatable()` since a liquidation is always a decrease position operation. A position is liquidable or not has nothing to do with exiting trade operations and only depend on the parameters of the position per se.

Current implementation has a problem for an increase order: Given a Increase order, for example, increase a position by $200, when `PositionUtils.validatePosition()` is called, which is after the position has been increased, we should not consider another $200 increase in `isPositionLiquidatable()` again as part of the price impact calculation. This is double-accouting for price impact calculation, one during the position increasing process, and another in the position validation process. On the other hand, if we use `false` here, then we are considering a decrease order (since a liquidation is a decrease order) and evaluate the hypothetical price impact if the position will be liquidated.

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/position/PositionUtils.sol#L304-L412

Our POC code confirms my finding: intially, we don't have any positions, after executing a LimitIncrease order, the priceImpactUsd is evaluaed as follows (notice initialDiffUsd = 0):

> PositionPricingUtils.getPriceImpactUsd started...
> openInterestParams.longOpenInterest: 0
> openInterestParams.shortOpenInterest: 0 initialDiffUsd: 0 nextDiffUsd: 112345670000000000000000000000000 positiveImpactFactor: 500000000000000000000 negativeImpactFactor: 1000000000000000000000 positiveImpactUsd: 0
> negativeImpactUsd: 6310774783874449910000 deltaDiffUsd: 6310774783874449910000 priceImpactUsd: -6310774783874449910000 PositionPricingUtils.getPriceImpactUsd() completed. Initial priceImpactUsd: -6310774783874449910000
> Capped priceImpactUsd: -6310774783874449910000

Then, during validation, when `PositionUtils.validatePosition()` is called, the double accouting occurs, notice the `nextDiffUsd` is doubled, as if the limitOrder was executed for another time!

> PositionPricingUtils.getPriceImpactUsd started...
> openInterestParams.longOpenInterest: 112345670000000000000000000000000
> openInterestParams.shortOpenInterest: 0 initialDiffUsd: 112345670000000000000000000000000 nextDiffUsd: 224691340000000000000000000000000 impactFactor: 1000000000000000000000 impactExponentFactor: 2000000000000000000000000000000 deltaDiffUsd: 189323243516233497450000 priceImpactUsd: -189323243516233497450000 priceImpactUsd: -189323243516233497450000 adjusted 2: priceImpactUsd: 0

The POC code is as follows, pay attention to the `testLimit()` and the execution of `createLimitIncreaseOrder()`. Please comment out the checks for signature, timestamp and block number for oracle price in the source code to run the testing smoothly without revert.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../contracts/role/RoleStore.sol";
import "../contracts/router/ExchangeRouter.sol";
import "../contracts/data/DataStore.sol";
import "../contracts/referral/ReferralStorage.sol";
```

```
import "../contracts/token/IWNT.sol";
import "../contracts/token/WNT.sol";
import "../contracts/token/SOLToken.sol";
import "../contracts/token/USDC.sol";
import "../contracts/token/tokenA.sol";
import "../contracts/token/tokenB.sol";
import "../contracts/token/tokenC.sol";

import "../contracts/market/MarketFactory.sol";
import "../contracts/deposit/DepositUtils.sol";
import "../contracts/oracle/OracleUtils.sol";
import "@openzeppelin/contracts/utils/introspection/ERC165Checker.sol";
import "../contracts/withdrawal/WithdrawalUtils.sol";
import "../contracts/order/Order.sol";
import "../contracts/order/BaseOrderUtils.sol";
import "../contracts/price/Price.sol";
import "../contracts/utils/Debug.sol";
import "../contracts/position/Position.sol";
import "../contracts/exchange/LiquidationHandler.sol";
import "../contracts/utils/Calc.sol";
import "@openzeppelin/contracts/utils/math/SignedMath.sol";
import "@openzeppelin/contracts/utils/math/SafeCast.sol";


contract CounterTest is Test, Debug{
     using SignedMath for int256;
    using SafeCast for uint256;


    WNT _wnt;
    USDC _usdc;
    SOLToken _sol;
    tokenA _tokenA;
    tokenB _tokenB;
    tokenC _tokenC;

    RoleStore _roleStore;
    Router _router;
    DataStore _dataStore;
    EventEmitter _eventEmitter;
    DepositVault _depositVault;
    OracleStore _oracleStore;
    Oracle _oracle;
    DepositHandler _depositHandler;
    WithdrawalVault _withdrawalVault;
    WithdrawalHandler _withdrawalHandler;
    OrderHandler _orderHandler;
```

```
    SwapHandler _swapHandler;
    LiquidationHandler _liquidationHandler;
    ReferralStorage _referralStorage;
    OrderVault _orderVault;
    ExchangeRouter _erouter;
    MarketFactory _marketFactory;
    Market.Props _marketProps1;
    Market.Props _marketPropsAB;
    Market.Props _marketPropsBC;
    Market.Props _marketPropsCwnt;


    address depositor1;
    address depositor2;
    address depositor3;
    address uiFeeReceiver = address(333);


    function testGetFundingAmountPerSizeDelta() public{
        uint result = MarketUtils.getFundingAmountPerSizeDelta(2e15, 1e15+1,
↪   true);
        console2.log("result: %d", result);
        uint256 correctResult = 2e15 * 1e15 * 1e30 + 1e15; // this is a real
↪   round up
        correctResult = correctResult/(1e15+1);
        console2.log("correctResult: %d", correctResult);
        assertTrue(result  == 1e15 * 1e30);
    }



    function setUp() public {
        _wnt = new WNT();
        _usdc = new USDC();
        _sol = new SOLToken();
        _tokenA = new tokenA();
        _tokenB = new tokenB();
         _tokenC = new tokenC();



         _roleStore = new RoleStore();
         _router = new Router(_roleStore);
         _dataStore = new DataStore(_roleStore);

         _eventEmitter= new EventEmitter(_roleStore);
        _depositVault = new DepositVault(_roleStore, _dataStore);
        _oracleStore = new OracleStore(_roleStore, _eventEmitter);
```


```
        _oracle = new Oracle(_roleStore, _oracleStore);
        console2.logString("_oracle:"); console2.logAddress(address(_oracle));

         _depositHandler = new DepositHandler(_roleStore, _dataStore,
↪  _eventEmitter, _depositVault, _oracle);
         console2.logString("_depositHandler:");
↪  console2.logAddress(address(_depositHandler));


       _withdrawalVault = new WithdrawalVault(_roleStore, _dataStore);
        _withdrawalHandler = new WithdrawalHandler(_roleStore, _dataStore,
↪  _eventEmitter, _withdrawalVault, _oracle);


        _swapHandler = new SwapHandler(_roleStore);
        _orderVault = new OrderVault(_roleStore, _dataStore);
        _referralStorage = new ReferralStorage();



        _orderHandler = new OrderHandler(_roleStore, _dataStore, _eventEmitter,
↪  _orderVault, _oracle, _swapHandler, _referralStorage);
        _erouter = new ExchangeRouter(_router, _roleStore, _dataStore,
↪  _eventEmitter, _depositHandler, _withdrawalHandler, _orderHandler);
         console2.logString("_erouter:"); console2.logAddress(address(_erouter));
         _liquidationHandler = new LiquidationHandler(_roleStore, _dataStore,
↪  _eventEmitter, _orderVault, _oracle, _swapHandler, _referralStorage);

        _referralStorage.setHandler(address(_orderHandler), true);

        /* set myself as the controller so that I can set the address of WNT
↪  (wrapped native token contracdt) */
        _roleStore.grantRole(address(this), Role.CONTROLLER);
        _roleStore.grantRole(address(this), Role.MARKET_KEEPER);

        _dataStore.setUint(Keys.MAX_SWAP_PATH_LENGTH, 5); // at most 5 markets
↪  in the path

        _dataStore.setAddress(Keys.WNT, address(_wnt));

        /* set the token transfer gas limit for wnt as 3200 */
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_wnt)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_usdc)), 32000);


        /* create a market (SQL, WNT, ETH, USDC) */
        _marketFactory = new MarketFactory(_roleStore, _dataStore,
↪  _eventEmitter);
```


```
        console2.logString("_marketFactory:");
↪   console2.logAddress(address(_marketFactory));
        _roleStore.grantRole(address(_marketFactory), Role.CONTROLLER); // to
↪   save a market's props
        _roleStore.grantRole(address(_erouter), Role.CONTROLLER);
        _roleStore.grantRole(address(_depositHandler), Role.CONTROLLER);
        _roleStore.grantRole(address(_withdrawalHandler), Role.CONTROLLER);
         _roleStore.grantRole(address(_swapHandler), Role.CONTROLLER);
        _roleStore.grantRole(address(_orderHandler), Role.CONTROLLER);
        _roleStore.grantRole(address(_liquidationHandler), Role.CONTROLLER);
        _roleStore.grantRole(address(_oracleStore), Role.CONTROLLER); // so it
↪   can call EventEmitter
        _roleStore.grantRole(address(_oracle), Role.CONTROLLER); // so it can
↪   call EventEmitter
        _roleStore.grantRole(address(this), Role.ORDER_KEEPER);
        _roleStore.grantRole(address(this), Role.LIQUIDATION_KEEPER);


        _marketProps1 = _marketFactory.createMarket(address(_sol),
↪   address(_wnt), address(_usdc), keccak256(abi.encode("sol-wnt-usdc")));
        _marketPropsAB = _marketFactory.createMarket(address(0),
↪   address(_tokenA), address(_tokenB),
↪   keccak256(abi.encode("swap-tokenA-tokenB")));
        _marketPropsBC = _marketFactory.createMarket(address(0),
↪   address(_tokenB), address(_tokenC),
↪   keccak256(abi.encode("swap-tokenB-tokenC")));
        _marketPropsCwnt = _marketFactory.createMarket(address(0),
↪   address(_tokenC), address(_wnt), keccak256(abi.encode("swap-tokenC-wnt")));


        _dataStore.setUint(Keys.minCollateralFactorForOpenInterestMultiplierKey(
↪   _marketProps1.marketToken, true), 1e25);
        _dataStore.setUint(Keys.minCollateralFactorForOpenInterestMultiplierKey(
↪   _marketProps1.marketToken, false), 1e25);

        // see fees for the market
        _dataStore.setUint(Keys.swapFeeFactorKey(_marketProps1.marketToken),
↪   0.05e30); // 5%
        _dataStore.setUint(Keys.SWAP_FEE_RECEIVER_FACTOR, 0.5e30);
        _dataStore.setUint(Keys.positionFeeFactorKey(_marketProps1.marketToken),
↪   0.00001234e30); // 2%
        _dataStore.setUint(Keys.POSITION_FEE_RECEIVER_FACTOR, 0.15e30);
         _dataStore.setUint(Keys.MAX_UI_FEE_FACTOR, 0.01e30);
        _dataStore.setUint(Keys.uiFeeFactorKey(uiFeeReceiver), 0.01e30); // only
↪   when this is set, one can receive ui fee, so stealing is not easy

↪   _dataStore.setInt(Keys.poolAmountAdjustmentKey(_marketProps1.marketToken,
↪   _marketProps1.longToken), 1);
```


```
↪  _dataStore.setInt(Keys.poolAmountAdjustmentKey(_marketProps1.marketToken,
↪  _marketProps1.shortToken), 1);
        _dataStore.setUint(Keys.swapImpactExponentFactorKey(_marketProps1.market
↪  Token), 10e28);
        _dataStore.setUint(Keys.swapImpactFactorKey(_marketProps1.marketToken,
↪  true), 0.99e30);
        _dataStore.setUint(Keys.swapImpactFactorKey(_marketProps1.marketToken,
↪  false), 0.99e30);




        // set gas limit to transfer a token
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_sol)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_wnt)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_usdc)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_tokenA)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_tokenB)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_tokenC)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_marketProps1.mark
↪  etToken)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_marketPropsAB.mar
↪  ketToken)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_marketPropsBC.mar
↪  ketToken)), 32000);
        _dataStore.setUint(Keys.tokenTransferGasLimit(address(_marketPropsCwnt.m
↪  arketToken)), 32000);


        /* Configure the system parameters/limits here */
        _dataStore.setUint(Keys.MAX_CALLBACK_GAS_LIMIT, 10000);
        _dataStore.setUint(Keys.EXECUTION_GAS_FEE_BASE_AMOUNT, 100);
        _dataStore.setUint(Keys.MAX_ORACLE_PRICE_AGE, 2 hours);
        _dataStore.setUint(Keys.MIN_ORACLE_BLOCK_CONFIRMATIONS, 3);
        _dataStore.setUint(Keys.MIN_COLLATERAL_USD, 1e30);  // just require $1
↪  as min collateral usd
        _dataStore.setUint(Keys.reserveFactorKey(_marketProps1.marketToken,
↪  true), 5e29); // 50%
        _dataStore.setUint(Keys.reserveFactorKey(_marketProps1.marketToken,
↪  false), 5e29);

↪  _dataStore.setUint(Keys.fundingExponentFactorKey(_marketProps1.marketToken),
↪  1.1e30); // 2 in 30 decimals like a square, cube, etc
         _dataStore.setUint(Keys.fundingFactorKey(_marketProps1.marketToken),
↪  0.0000001e30);
         _dataStore.setUint(Keys.borrowingFactorKey(_marketProps1.marketToken,
↪  true), 0.87e30);
```


```
        _dataStore.setUint(Keys.borrowingFactorKey(_marketProps1.marketToken,
↪  false), 0.96e30);
        _dataStore.setUint(Keys.borrowingExponentFactorKey(_marketProps1.market
↪  Token, true), 2.1e30);
        _dataStore.setUint(Keys.borrowingExponentFactorKey(_marketProps1.market
↪  Token, false), 2.3e30);
        _dataStore.setUint(Keys.positionImpactExponentFactorKey(_marketProps1.m
↪  arketToken), 2e30);

↪  _dataStore.setUint(Keys.positionImpactFactorKey(_marketProps1.marketToken,
↪  true), 5e22);

↪  _dataStore.setUint(Keys.positionImpactFactorKey(_marketProps1.marketToken,
↪  false), 1e23);

        // set the limit of market tokens

        _dataStore.setUint(Keys.maxPoolAmountKey(_marketProps1.marketToken,
↪  _marketProps1.longToken), 1000e18);
        _dataStore.setUint(Keys.maxPoolAmountKey(_marketProps1.marketToken,
↪  _marketProps1.shortToken), 1000e18);
        _dataStore.setUint(Keys.maxPoolAmountKey(_marketPropsAB.marketToken,
↪  _marketPropsAB.longToken), 1000e18);
        _dataStore.setUint(Keys.maxPoolAmountKey(_marketPropsAB.marketToken,
↪  _marketPropsAB.shortToken), 1000e18);
        _dataStore.setUint(Keys.maxPoolAmountKey(_marketPropsBC.marketToken,
↪  _marketPropsBC.longToken), 1000e18);
        _dataStore.setUint(Keys.maxPoolAmountKey(_marketPropsBC.marketToken,
↪  _marketPropsBC.shortToken), 1000e18);
        _dataStore.setUint(Keys.maxPoolAmountKey(_marketPropsCwnt.marketToken,
↪  _marketPropsCwnt.longToken), 1000e18);
        _dataStore.setUint(Keys.maxPoolAmountKey(_marketPropsCwnt.marketToken,
↪  _marketPropsCwnt.shortToken), 1000e18);


        // set max open interest for each market
        _dataStore.setUint(Keys.maxOpenInterestKey(_marketProps1.marketToken,
↪  true), 1e39); // 1B $
        _dataStore.setUint(Keys.maxOpenInterestKey(_marketProps1.marketToken,
↪  false), 1e39); // 1B $


↪  _dataStore.setUint(Keys.maxPnlFactorKey(Keys.MAX_PNL_FACTOR_FOR_WITHDRAWALS,
↪  _marketProps1.marketToken, true), 10**29); // maxPnlFactor = 10% for long

↪  _dataStore.setUint(Keys.maxPnlFactorKey(Keys.MAX_PNL_FACTOR_FOR_WITHDRAWALS,
↪  _marketProps1.marketToken, false), 10**29); // maxPnlFactor = 10% for short
```


```
        // _dataStore.setBool(Keys.cancelDepositFeatureDisabledKey(address(_depo
↪   sitHandler)), true);
        _dataStore.setBool(Keys.cancelOrderFeatureDisabledKey(address(_orderHand
↪   ler), uint256(Order.OrderType.MarketIncrease)), true);

        addFourSigners();
        address(_wnt).call{value: 10000e18}("");
        depositor1 = address(0x801);
        depositor2 = address(0x802);
        depositor3 = address(0x803);

        // make sure each depositor has some tokens.
        _wnt.transfer(depositor1, 1000e18);
        _wnt.transfer(depositor2, 1000e18);
        _wnt.transfer(depositor3, 1000e18);
        _usdc.transfer(depositor1, 1000e18);
        _usdc.transfer(depositor2, 1000e18);
        _usdc.transfer(depositor3, 1000e18);
        _tokenA.transfer(depositor1, 1000e18);
        _tokenB.transfer(depositor1, 1000e18);
        _tokenC.transfer(depositor1, 1000e18);

        printAllTokens();
    }

    error Unauthorized(string);
   // error Error(string);


function testLimit() public{
         OracleUtils.SetPricesParams memory priceParams =
↪   createSetPricesParams();

         vm.roll(block.number+2); // block 3


         bytes32 key = createDepositNoSwap(_marketProps1, depositor1, 90e18,
↪   true); // create a deposit at block 3 which is within range (2, 6)
         _depositHandler.executeDeposit(key, priceParams);
         uint mintedMarketTokens =
↪   IERC20(_marketProps1.marketToken).balanceOf(depositor1);
         key = createDepositNoSwap(_marketProps1, depositor1, 100e18, false);
↪   // create a deposit at block 3 which is within range (2, 6)
         _depositHandler.executeDeposit(key, priceParams);
         mintedMarketTokens =
↪   IERC20(_marketProps1.marketToken).balanceOf(depositor1);
         console2.log("Experiment 1  is completed.");
```


```
        // console2.log("PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP");

        key = createMarketSwapOrder(depositor1,  address(_wnt), 1e15); //
↪  create a deposit at block 3 which is within range (2, 6)
        _orderHandler.executeOrder(key, priceParams);
        console2.log("Experiment 2  is completed.");


      console2.log("\n\n depositor 1 createMarketIncreaseOrder");
       key = createMarketIncreaseOrder(depositor1, _marketProps1.marketToken,
↪  _marketProps1.longToken, 20e18, 1001e30, 106000000000000, true); //
       console2.log("\nExecuting the order...");
       _orderHandler.executeOrder(key, priceParams);
       Position.printPosition(_dataStore, depositor1,
↪  _marketProps1.marketToken, _marketProps1.longToken, true);
       console2.log("Experiment 3  is completed.");



      console2.log("\n\n depositor 2 createMarketIncreaseOrder");
       key = createMarketIncreaseOrder(depositor2, _marketProps1.marketToken,
↪  _marketProps1.longToken, 110e18, 13e30, 101000000000000, false); // 110 usdc
↪  as collateral
       console2.log("\nExecuting the order...");
       _orderHandler.executeOrder(key, priceParams);
       Position.printPosition(_dataStore, depositor2,
↪  _marketProps1.marketToken, _marketProps1.longToken, false);
     console2.log("Experiment 4  is completed.");



      console2.log("PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP");
      vm.warp(2 days);
      setIndexTokenPrice(priceParams, 98, 100); // send 20e18 USDC, increase
↪  $13.123 in a long position with trigger price 101
      key = createLimitIncreaseOrder(depositor3, _marketProps1.marketToken,
↪  _marketProps1.shortToken, 23e18, 1.1234567e30, 101000000000000, true); //
↪  collateral token, usdsize, price
      console2.log("a LimitIncrease order created by depositor3 with key: ");
       console2.logBytes32(key);
       Position.printPosition(_dataStore, depositor3,
↪  _marketProps1.marketToken, _marketProps1.shortToken, true);
       console2.log("\n\nExecuting the order, exiting moment...\n\n");
       _orderHandler.executeOrder(key, priceParams);
       Position.printPosition(_dataStore, depositor3,
↪  _marketProps1.marketToken, _marketProps1.shortToken, true);
       console2.log("Experiment  5 is completed.\n");
```

```
        // depositor3 creates a LimitDecrease order
        /*
        setIndexTokenPrice(priceParams, 120, 125);
        key = createLimitDecreaseOrder(depositor3, _marketProps1.marketToken,
↪   _marketProps1.shortToken, 7e18, 58e30, 120000000000000, 120000000000000,
↪   true); // retrieve $50? collateral token, usdsize, acceptible price
        console2.log("a LimitIncrease order created by depositor3 with key: ");
         console2.logBytes32(key);
         Position.printPosition(_dataStore, depositor3,
↪   _marketProps1.marketToken, _marketProps1.shortToken, true);
         console2.log("\n\nExecuting the order, exiting moment...\n\n");
          _orderHandler.executeOrder(key, priceParams);
        console2.log("Experiment 7 for is completed.");
        */
}

function testMarketDecrease() public{

          OracleUtils.SetPricesParams memory priceParams =
↪   createSetPricesParams();

          vm.roll(block.number+2); // block 3


          bytes32 key = createDepositNoSwap(_marketProps1, depositor1, 90e18,
↪   true); // create a deposit at block 3 which is within range (2, 6)
          _depositHandler.executeDeposit(key, priceParams);
          uint mintedMarketTokens =
↪   IERC20(_marketProps1.marketToken).balanceOf(depositor1);
          key = createDepositNoSwap(_marketProps1, depositor1, 100e18, false);
↪   // create a deposit at block 3 which is within range (2, 6)
          _depositHandler.executeDeposit(key, priceParams);
          mintedMarketTokens =
↪   IERC20(_marketProps1.marketToken).balanceOf(depositor1);
         console2.log("Experiment 1  is completed.");




          console2.log("\n\n depositor 2 deposit into marketProps1");
          key = createDepositNoSwap(_marketProps1, depositor2, 100e18, true);
          _depositHandler.executeDeposit(key, priceParams);
         mintedMarketTokens =
↪   IERC20(_marketProps1.marketToken).balanceOf(depositor2);
          printPoolsAmounts();
         console2.log("Experiment 2  is completed.");
```

```
        console2.log("\n\n depositor 1 createMarketIncreaseOrder");
         key = createMarketIncreaseOrder(depositor1, _marketProps1.marketToken,
↪   _marketProps1.longToken, 20e18, 1e25, 106000000000000, true); //
         console2.log("\nExecuting the order...");
         _orderHandler.executeOrder(key, priceParams);
         Position.printPosition(_dataStore, depositor1,
↪   _marketProps1.marketToken, _marketProps1.longToken, true);
         console2.log("Experiment 3  is completed.");



        console2.log("\n\n depositor 2 createMarketIncreaseOrder");
         key = createMarketIncreaseOrder(depositor2, _marketProps1.marketToken,
↪   _marketProps1.longToken, 110e18, 1e25, 101000000000000, false); // 110 usdc
↪   as collateral
         console2.log("\nExecuting the order...");
         _orderHandler.executeOrder(key, priceParams);
         Position.printPosition(_dataStore, depositor2,
↪   _marketProps1.marketToken, _marketProps1.longToken, false);
       console2.log("Experiment 4  is completed.");

        console2.log("*****************************************");

         // deposit 2 will execute a marketDecreaseOrder now
         key =  createMarketDecreaseOrder(depositor2, _marketProps1.marketToken,
↪   _marketProps1.longToken, 70000000000000, 5e23, false) ; // decrease by 5%
         console2.log("a market desced order created with key: ");
         console2.logBytes32(key);
         console2.log("\nExecuting the order...");
         setIndexTokenPrice(priceParams, 60, 65); // we have a profit for a
↪   short position
         _orderHandler.executeOrder(key, priceParams);
         Position.printPosition(_dataStore, depositor2,
↪   _marketProps1.marketToken, _marketProps1.longToken, false);
         console2.log("Experiment 5  is completed.");

          printAllTokens();
}



function  testLiquidation() public{
          // blockrange (2, 6)
          OracleUtils.SetPricesParams memory priceParams =
↪   createSetPricesParams();

          vm.roll(block.number+2); // block 3
```

```
        bytes32 key = createDepositNoSwap(_marketProps1, depositor1, 90e18,
↪  true); // create a deposit at block 3 which is within range (2, 6)
        _depositHandler.executeDeposit(key, priceParams);
        uint mintedMarketTokens =
↪  IERC20(_marketProps1.marketToken).balanceOf(depositor1);
        key = createDepositNoSwap(_marketProps1, depositor1, 100e18, false);
↪  // create a deposit at block 3 which is within range (2, 6)
        _depositHandler.executeDeposit(key, priceParams);
        mintedMarketTokens =
↪  IERC20(_marketProps1.marketToken).balanceOf(depositor1);
      console2.log("Experiment 1  is completed.");




        console2.log("\n\n depositor 2 deposit into marketProps1");
        key = createDepositNoSwap(_marketProps1, depositor2, 100e18, true);
        _depositHandler.executeDeposit(key, priceParams);
      mintedMarketTokens =
↪  IERC20(_marketProps1.marketToken).balanceOf(depositor2);
        printPoolsAmounts();
      console2.log("Experiment 2  is completed.");


     console2.log("\n\n depositor 1 createMarketIncreaseOrder");
      key = createMarketIncreaseOrder(depositor1, _marketProps1.marketToken,
↪  _marketProps1.longToken, 10e18, 1e25, 106000000000000, true);
      console2.log("\nExecuting the order...");
      _orderHandler.executeOrder(key, priceParams);
      Position.printPosition(_dataStore, depositor1,
↪  _marketProps1.marketToken, _marketProps1.longToken, true);
      console2.log("Experiment 3  is completed.");



     console2.log("\n\n depositor 2 createMarketIncreaseOrder");
      key = createMarketIncreaseOrder(depositor2, _marketProps1.marketToken,
↪  _marketProps1.shortToken, 100e18, 1e25, 101000000000000, false);
      console2.log("\nExecuting the order...");
      _orderHandler.executeOrder(key, priceParams);
      Position.printPosition(_dataStore, depositor2,
↪  _marketProps1.marketToken, _marketProps1.shortToken, false);
    console2.log("Experiment 4  is completed.");
```


```
        // deposit 2 will execute a marketDecreaseOrder now
        key =  createMarketDecreaseOrder(depositor2, _marketProps1.marketToken,
↪  _marketProps1.shortToken, 106000000000000, 5e23, false) ; // decrease by 5%
        console2.log("a market desced order created with key: ");
        console2.logBytes32(key);
        console2.log("\nExecuting the order...");
        setIndexTokenPrice(priceParams, 84, 90);
        _orderHandler.executeOrder(key, priceParams);
        Position.printPosition(_dataStore, depositor2,
↪  _marketProps1.marketToken, _marketProps1.shortToken, false);
        console2.log("Experiment 5  is completed.");




       // depositor3 will execute a LimitIncrease Order now
       key = createMarketIncreaseOrder(depositor3, _marketProps1.marketToken,
↪  _marketProps1.shortToken, 20e18, 200e30, 101000000000000, true); //
↪  collateral token, usdsize, price
       console2.log("a LimitIncrease order created by depositor3 with key: ");
        console2.logBytes32(key);
        Position.printPosition(_dataStore, depositor3,
↪  _marketProps1.marketToken, _marketProps1.shortToken, true);
        console2.log("\n\nExecuting the order, exiting moment...\n\n");
        _orderHandler.executeOrder(key, priceParams);
        Position.printPosition(_dataStore, depositor3,
↪  _marketProps1.marketToken, _marketProps1.shortToken, true);
        console2.log("Experiment  6 is completed.\n");


       // depositor3 creates a LimitDecrease order
       setIndexTokenPrice(priceParams, 120, 125);
       key = createLimitDecreaseOrder(depositor3, _marketProps1.marketToken,
↪  _marketProps1.shortToken, 7e18, 58e30, 120000000000000, 120000000000000,
↪  true); // retrieve $50? collateral token, usdsize, acceptible price
       console2.log("a LimitIncrease order created by depositor3 with key: ");
        console2.logBytes32(key);
        Position.printPosition(_dataStore, depositor3,
↪  _marketProps1.marketToken, _marketProps1.shortToken, true);
        console2.log("\n\nExecuting the order, exiting moment...\n\n");
         _orderHandler.executeOrder(key, priceParams);
       console2.log("Experiment 7 for is completed.");

       // depositor3 creates a stopLossDecrease order
       setIndexTokenPrice(priceParams, 97, 99);
       key = createStopLossDecrease(depositor3, _marketProps1.marketToken,
↪  _marketProps1.shortToken, 7e18, 58e30, 95000000000000, 92000000000000,
↪  true); // retrieve $50? collateral token, usdsize, acceptible price
```


```
        console2.log("a StopLossDecrease order created by depositor3 with key:
↪  ");
         console2.logBytes32(key);
        // Position.printPosition(_dataStore, depositor3,
↪  _marketProps1.marketToken, _marketProps1.shortToken, true);

        console2.log("\n\nExecuting the order, exiting moment...\n\n");
        _orderHandler.executeOrder(key, priceParams);
        console2.log("Experiment 8 is completed.");



↪  console2.log("\n\n*********************************************\n\n");


        // depositor3 creates a Liquidation order
        setIndexTokenPrice(priceParams, 75, 75);
        console2.log("Liquidate a position...");
        Position.printPosition(_dataStore, depositor3,
↪  _marketProps1.marketToken, _marketProps1.shortToken, true);
        _liquidationHandler.executeLiquidation(depositor3,
↪  _marketProps1.marketToken, _marketProps1.shortToken, true, priceParams);
        console2.log("Experiment 9 is completed.");


         // printPoolsAmounts();
         printAllTokens();




}

function printAllTokens() startedCompleted("printAllTokens") public
{
      console2.log("\nTokens used in this test:");
        console2.log("_wnt: "); console2.logAddress(address(_wnt));
        console2.log("_usdc: "); console2.logAddress(address(_usdc));
        console2.log("_sol: "); console2.logAddress(address(_sol));
        console2.log("_tokenA: "); console2.logAddress(address(_tokenA));
        console2.log("_tokenB: "); console2.logAddress(address(_tokenB));
        console2.log("_tokenC: "); console2.logAddress(address(_tokenC));
        console2.logString("test contract address:");
↪  console2.logAddress(address(this));

        console2.log("_marketProps1 market token: ");
↪  console2.logAddress(address(_marketProps1.marketToken));
```

```
        console2.log("_marketPropsAB market token: ");
↪   console2.logAddress(address(_marketPropsAB.marketToken));
        console2.log("_marketPropsBC market token: ");
↪   console2.logAddress(address(_marketPropsBC.marketToken));
        console2.log("_marketProps1Cwnt market token: ");
↪   console2.logAddress(address(_marketPropsCwnt.marketToken));
        console2.log("\n");


}


function printMarketTokenAmount() public
{   console2.log("Market token address: ");
    console2.logAddress(address(_marketProps1.marketToken));
    console2.log("depositor1 market token amount: %d",
↪   IERC20(_marketProps1.marketToken).balanceOf(depositor1));
    console2.log("depositor2 market token amount: %d",
↪   IERC20(_marketProps1.marketToken).balanceOf(depositor2));
    console2.log("depositor3 market token amount: %d",
↪   IERC20(_marketProps1.marketToken).balanceOf(depositor3));
}

function printLongShortTokens(address account) public
{
    console2.log("balance for "); console2.logAddress(account);
    console2.log("_wnt balance:", _wnt.balanceOf(account));
    console2.log("usdc balance:", _usdc.balanceOf(account));
}




function addFourSigners() private {
    _oracleStore.addSigner(address(901));
    _oracleStore.addSigner(address(902));
    _oracleStore.addSigner(address(903));
    _oracleStore.addSigner(address(904));
}


function setIndexTokenPrice(OracleUtils.SetPricesParams memory priceParams,
↪   uint256 minP, uint256 maxP) public
{
    uint256 mask1 = ~uint256(type(uint96).max);   // (32*3 of 1's)
    console2.logBytes32(bytes32(mask1));

    uint256 minPrice = minP;
```

```
    minPrice = minPrice << 32 | minP;
    minPrice = minPrice << 32 | minP;

    uint256 maxPrice = maxP;
    maxPrice = maxPrice << 32 | maxP;
    maxPrice = maxPrice << 32 | maxP;

    priceParams.compactedMinPrices[0] = (priceParams.compactedMinPrices[0] &
↪   mask1) | minPrice;
    priceParams.compactedMaxPrices[0] = (priceParams.compactedMaxPrices[0] &
↪   mask1) | maxPrice;
}


function createSetPricesParams() public returns (OracleUtils.SetPricesParams
↪   memory) {
          uint256 signerInfo = 3;    // signer 904
          signerInfo = signerInfo << 16 | 2; // signer 903
          signerInfo = signerInfo << 16 | 1; // signer 902
          signerInfo = signerInfo << 16 | 3; // number of singers
          // will read out as 902, 903, 904 from the lowest first

          // the number of tokens, 6
          address[] memory tokens = new address[](6);
          tokens[0] = address(_sol);
          tokens[1] = address(_wnt);
          tokens[2] = address(_usdc);
          tokens[3] = address(_tokenA);
          tokens[4] = address(_tokenB);
          tokens[5] = address(_tokenC);

         // must be equal to the number of tokens 6, 64 for each one, so 64*6.
↪   64*4 for one element, so need two elements
          uint256[] memory compactedMinOracleBlockNumbers = new uint256[](2);
          compactedMinOracleBlockNumbers[0] = block.number+1;
          compactedMinOracleBlockNumbers[0] = compactedMinOracleBlockNumbers[0]
↪   << 64 | block.number+1;
          compactedMinOracleBlockNumbers[0] = compactedMinOracleBlockNumbers[0]
↪   << 64 | block.number+1;
          compactedMinOracleBlockNumbers[0] = compactedMinOracleBlockNumbers[0]
↪   << 64 | block.number+1;

          compactedMinOracleBlockNumbers[1] = block.number+1;
          compactedMinOracleBlockNumbers[1] = compactedMinOracleBlockNumbers[0]
↪   << 64 | block.number+1;

        // must be equal to the number of tokens 6, 64 for each one, so 64*6.
↪   64*4 for one element, so need two elements
```


```
        uint256[] memory compactedMaxOracleBlockNumbers = new uint256[](2);
        compactedMaxOracleBlockNumbers[0] = block.number+5;
        compactedMaxOracleBlockNumbers[0] = compactedMaxOracleBlockNumbers[0]
↪  << 64 | block.number+5;
        compactedMaxOracleBlockNumbers[0] = compactedMaxOracleBlockNumbers[0]
↪  << 64 | block.number+5;
        compactedMaxOracleBlockNumbers[0] = compactedMaxOracleBlockNumbers[0]
↪  << 64 | block.number+5;

        compactedMaxOracleBlockNumbers[1] = block.number+5;
        compactedMaxOracleBlockNumbers[1] = compactedMaxOracleBlockNumbers[0]
↪  << 64 | block.number+5;

       // must be equal to the number of tokens 6, 64 for each one, so 64*6.
↪  64*4 for one element, so need two elements
        uint256[] memory compactedOracleTimestamps = new uint256[](2);
        compactedOracleTimestamps[0]  =  9;
        compactedOracleTimestamps[0] = compactedOracleTimestamps[0] << 64 | 8;
        compactedOracleTimestamps[0] = compactedOracleTimestamps[0] << 64 | 7;
        compactedOracleTimestamps[0] = compactedOracleTimestamps[0] << 64 | 7;

        compactedOracleTimestamps[1]  =  9;
        compactedOracleTimestamps[1] = compactedOracleTimestamps[0] << 64 | 8;


         // must be equal to the number of tokens, 8 for each, so 8*6= 48,
↪  only need one element
         uint256[] memory compactedDecimals = new uint256[](1);
         compactedDecimals[0] =  12;
         compactedDecimals[0] =  compactedDecimals[0] << 8 | 12;
         compactedDecimals[0] =  compactedDecimals[0] << 8 | 12;
         compactedDecimals[0] =  compactedDecimals[0] << 8 | 12;
         compactedDecimals[0] =  compactedDecimals[0] << 8 | 12;
         compactedDecimals[0] =  compactedDecimals[0] << 8 | 12;


         // three signers, 6 tokens, so we have 3*6 = 18 entries, each entry
↪  takes 32 bits, so each 8 entries takes one element, we need 3 elements
         // price table:
         //      SOL:        100 101 102
         //      wnt:        200 201 203
         //      USDC        1   1    1
         //      tokenA      100 101  102
         //      tokenB      200 202  204
         //      tokenC      400 404  408

         uint256[] memory compactedMinPrices = new uint256[](3);
```

```
        compactedMinPrices[2] = 408;
        compactedMinPrices[2] = compactedMinPrices[2] << 32 | 404;

        compactedMinPrices[1] = 400;
        compactedMinPrices[1] = compactedMinPrices[1] << 32 | 204;
        compactedMinPrices[1] = compactedMinPrices[1] << 32 | 202;
        compactedMinPrices[1] = compactedMinPrices[1] << 32 | 200;
        compactedMinPrices[1] = compactedMinPrices[1] << 32 | 102;
        compactedMinPrices[1] = compactedMinPrices[1] << 32 | 101;
        compactedMinPrices[1] = compactedMinPrices[1] << 32 | 100;
        compactedMinPrices[1] = compactedMinPrices[1] << 32 | 1;

        compactedMinPrices[0] = 1;
        compactedMinPrices[0] = compactedMinPrices[0] << 32 | 1;
        compactedMinPrices[0] = compactedMinPrices[0] << 32 | 203;
        compactedMinPrices[0] = compactedMinPrices[0] << 32 | 201;
        compactedMinPrices[0] = compactedMinPrices[0] << 32 | 200;
        compactedMinPrices[0] = compactedMinPrices[0] << 32 | 102;
        compactedMinPrices[0] = compactedMinPrices[0] << 32 | 101;
        compactedMinPrices[0] = compactedMinPrices[0] << 32 | 100;

        // three signers, 6 tokens, so we have 3*6 = 18 entries, each entry
↪   takes 8 bits, so we just need one element

        uint256[] memory compactedMinPricesIndexes = new uint256[](1);
        compactedMinPricesIndexes[0] = 1;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 2;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 0;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 1;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 2;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 0;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 1;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 2;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 0;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 1;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 2;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 0;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 1;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 2;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 0;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 1;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 2;
        compactedMinPricesIndexes[0] = compactedMinPricesIndexes[0] << 8 | 0;

        // three signers, 6 tokens, so we have 3*6 = 18 entries, each entry
↪   takes 32 bits, so each 8 entries takes one element, we need 3 elements
        // price table:
        //      SOL:        105 106 107
```

```
        //      wnt:        205 206 208
        //      USDC        1   1    1
        //      tokenA      105 106  107
        //      tokenB      205 207  209
        //      tokenC      405 409  413
        uint256[] memory compactedMaxPrices = new uint256[](3);
        compactedMaxPrices[2] = 413;
        compactedMaxPrices[2] = compactedMaxPrices[2] << 32 | 409;

        compactedMaxPrices[1] = 405;
        compactedMaxPrices[1] = compactedMaxPrices[1] << 32 | 209;
        compactedMaxPrices[1] = compactedMaxPrices[1] << 32 | 207;
        compactedMaxPrices[1] = compactedMaxPrices[1] << 32 | 205;
        compactedMaxPrices[1] = compactedMaxPrices[1] << 32 | 107;
        compactedMaxPrices[1] = compactedMaxPrices[1] << 32 | 106;
        compactedMaxPrices[1] = compactedMaxPrices[1] << 32 | 105;
        compactedMaxPrices[1] = compactedMaxPrices[1] << 32 | 1;

        compactedMaxPrices[0] = 1;
        compactedMaxPrices[0] = compactedMaxPrices[0] << 32 | 1;
        compactedMaxPrices[0] = compactedMaxPrices[0] << 32 | 208;
        compactedMaxPrices[0] = compactedMaxPrices[0] << 32 | 206;
        compactedMaxPrices[0] = compactedMaxPrices[0] << 32 | 205;
        compactedMaxPrices[0] = compactedMaxPrices[0] << 32 | 107;
        compactedMaxPrices[0] = compactedMaxPrices[0] << 32 | 106;
        compactedMaxPrices[0] = compactedMaxPrices[0] << 32 | 105;


         // three signers, 6 tokens, so we have 3*6 = 18 entries, each entry
↪  takes 8 bits, so we just need one element

         uint256[] memory compactedMaxPricesIndexes = new uint256[](1);
         compactedMaxPricesIndexes[0] = 1;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 2;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 0;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 1;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 2;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 0;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 1;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 2;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 0;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 1;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 2;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 0;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 1;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 2;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 0;
         compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 1;
```

```
            compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 2;
            compactedMaxPricesIndexes[0] = compactedMaxPricesIndexes[0] << 8 | 0;

           // 3 signers and 6 tokens, so we need 3*6 signatures
            bytes[] memory signatures = new bytes[](18);
            for(uint i; i<18; i++){
                signatures[i] = abi.encode("SIGNATURE");
            }
            address[] memory priceFeedTokens;

        OracleUtils.SetPricesParams memory priceParams =
↪   OracleUtils.SetPricesParams(
                signerInfo,
                tokens,
                compactedMinOracleBlockNumbers,
                compactedMaxOracleBlockNumbers,
                compactedOracleTimestamps,
                compactedDecimals,
                compactedMinPrices,
                compactedMinPricesIndexes,
                compactedMaxPrices,
                compactedMaxPricesIndexes,
                signatures,
                priceFeedTokens
        );
        return priceParams;
}

/*
*  The current index token price (85, 90), a trader sets a trigger price to 100
↪   and then acceptabiel price to 95.
*  He like to long the index token.
*  1. Pick the primary price 90 since we long, so choose the max
*  2. Make sure 90 < 100, and pick (90, 100) as the custom price since we long
*  3. Choose price 95 since 95 is within the range, and it is the highest
↪   acceptible price. Choosing 90
*      will be in favor of the trader
*
*/

function createMarketSwapOrder(address account,  address inputToken, uint256
↪   inAmount) public returns(bytes32)
{
    address[] memory swapPath = new address[](1);
    swapPath[0] =  _marketProps1.marketToken;
    // swapPath[0] = _marketPropsAB.marketToken;
    // swapPath[1] = _marketPropsBC.marketToken;
    // swapPath[2] = _marketPropsCwnt.marketToken;
```

```
    vm.prank(account);
    _wnt.transfer(address(_orderVault), 3200);  // execution fee


    BaseOrderUtils.CreateOrderParams memory params;
    params.addresses.receiver = account;            // the account is the
↪   receiver
    params.addresses.callbackContract = address(0);
    params.addresses.uiFeeReceiver = account;   // set myself as the ui receiver
    // params.addresses.market = marketToken;
    params.addresses.initialCollateralToken = inputToken; // initial token
    params.addresses.swapPath = swapPath;

    // params.numbers.sizeDeltaUsd = sizeDeltaUsd;
    params.numbers.initialCollateralDeltaAmount = inAmount ; // this is actually
↪   useless, will be overidden by real transfer amount
     vm.prank(account);
    IERC20(inputToken).transfer(address(_orderVault), inAmount);  // this is the
↪   real amount


    // params.numbers.triggerPrice = triggerPrice;
    // params.numbers.acceptablePrice = acceptablePrice; // I can buy with this
↪   price or lower effective spread control
    params.numbers.executionFee = 3200;
    params.numbers.callbackGasLimit = 3200;
   // params.numbers.initialCollateralDeltaAmount = inAmount;
    params.numbers.minOutputAmount = 100; // use the control the final
↪   collateral amount, not for the position size delta, which is indirectly
↪   controlled by acceptable price

    params.orderType = Order.OrderType.MarketSwap;
    params.decreasePositionSwapType = Order.DecreasePositionSwapType.NoSwap;
    // params.isLong = isLong;
    params.shouldUnwrapNativeToken = false;
    params.referralCode = keccak256(abi.encode("MY REFERRAL"));

    vm.prank(account);
    bytes32 key = _erouter.createOrder(params);
    return key;
}
```

```
function createLiquidationOrder(address account, address marketToken, address
↪  collateralToken, uint256 collateralAmount, uint sizeDeltaUsd, uint
↪  triggerPrice, uint256 acceptablePrice, bool isLong) public returns(bytes32)
{
    address[] memory swapPath;

    //address[] memory swapPath = new address[](3);
    //swapPath[0] = _marketPropsAB.marketToken;
    //swapPath[1] = _marketPropsBC.marketToken;
    //swapPath[2] = _marketPropsCwnt.marketToken;


    vm.prank(account);
    _wnt.transfer(address(_orderVault), 3200);  // execution fee


    BaseOrderUtils.CreateOrderParams memory params;
    params.addresses.receiver = account;
    params.addresses.callbackContract = address(0);
    params.addresses.uiFeeReceiver = uiFeeReceiver;
    params.addresses.market = marketToken; // final market
    params.addresses.initialCollateralToken = collateralToken; // initial token
    params.addresses.swapPath = swapPath;

    params.numbers.sizeDeltaUsd = sizeDeltaUsd;
    // params.numbers.initialCollateralDeltaAmount = ; // this is actually
↪  useless, will be overidden by real transfer amount
     vm.prank(account);
    IERC20(collateralToken).transfer(address(_orderVault), collateralAmount);
↪  // this is the real amount


    params.numbers.triggerPrice = triggerPrice;
    params.numbers.acceptablePrice = acceptablePrice; // I can buy with this
↪  price or lower effective spread control
    params.numbers.executionFee = 3200;
    params.numbers.callbackGasLimit = 3200;
    params.numbers.minOutputAmount = 100; // use the control the final
↪  collateral amount, not for the position size delta, which is indirectly
↪  controlled by acceptable price

    params.orderType = Order.OrderType.Liquidation;
    params.decreasePositionSwapType = Order.DecreasePositionSwapType.NoSwap;
    params.isLong = isLong;
    params.shouldUnwrapNativeToken = false;
    params.referralCode = keccak256(abi.encode("MY REFERRAL"));

    vm.prank(account);
```

```
    bytes32 key = _erouter.createOrder(params);
    return key;
}



function createStopLossDecrease(address account, address marketToken, address
↪  collateralToken, uint256 collateralAmount, uint sizeDeltaUsd, uint
↪  triggerPrice, uint256 acceptablePrice, bool isLong) public returns(bytes32)
{
    address[] memory swapPath;

    //address[] memory swapPath = new address[](3);
    //swapPath[0] = _marketPropsAB.marketToken;
    //swapPath[1] = _marketPropsBC.marketToken;
    //swapPath[2] = _marketPropsCwnt.marketToken;


    vm.prank(account);
    _wnt.transfer(address(_orderVault), 3200);  // execution fee


    BaseOrderUtils.CreateOrderParams memory params;
    params.addresses.receiver = account;
    params.addresses.callbackContract = address(0);
    params.addresses.uiFeeReceiver = uiFeeReceiver;
    params.addresses.market = marketToken; // final market
    params.addresses.initialCollateralToken = collateralToken; // initial token
    params.addresses.swapPath = swapPath;

    params.numbers.sizeDeltaUsd = sizeDeltaUsd;
    // params.numbers.initialCollateralDeltaAmount = ; // this is actually
↪  useless, will be overidden by real transfer amount
     vm.prank(account);
    IERC20(collateralToken).transfer(address(_orderVault), collateralAmount);
↪  // this is the real amount


    params.numbers.triggerPrice = triggerPrice;
    params.numbers.acceptablePrice = acceptablePrice; // I can buy with this
↪  price or lower effective spread control
    params.numbers.executionFee = 3200;
    params.numbers.callbackGasLimit = 3200;
    params.numbers.minOutputAmount = 100; // use the control the final
↪  collateral amount, not for the position size delta, which is indirectly
↪  controlled by acceptable price

    params.orderType = Order.OrderType.StopLossDecrease;
```


```
    params.decreasePositionSwapType = Order.DecreasePositionSwapType.NoSwap;
    params.isLong = isLong;
    params.shouldUnwrapNativeToken = false;
    params.referralCode = keccak256(abi.encode("MY REFERRAL"));

    vm.prank(account);
    bytes32 key = _erouter.createOrder(params);
    return key;
}



function createLimitDecreaseOrder(address account, address marketToken, address
↪  collateralToken, uint256 collateralAmount, uint sizeDeltaUsd, uint
↪  triggerPrice, uint256 acceptablePrice, bool isLong) public returns(bytes32)
{
    address[] memory swapPath;

    //address[] memory swapPath = new address[](3);
    //swapPath[0] = _marketPropsAB.marketToken;
    //swapPath[1] = _marketPropsBC.marketToken;
    //swapPath[2] = _marketPropsCwnt.marketToken;


    vm.prank(account);
    _wnt.transfer(address(_orderVault), 3200);  // execution fee


    BaseOrderUtils.CreateOrderParams memory params;
    params.addresses.receiver = account;
    params.addresses.callbackContract = address(0);
    params.addresses.uiFeeReceiver = uiFeeReceiver;
    params.addresses.market = marketToken; // final market
    params.addresses.initialCollateralToken = collateralToken; // initial token
    params.addresses.swapPath = swapPath;

    params.numbers.sizeDeltaUsd = sizeDeltaUsd;
    // params.numbers.initialCollateralDeltaAmount = ; // this is actually
↪  useless, will be overidden by real transfer amount
     vm.prank(account);
    IERC20(collateralToken).transfer(address(_orderVault), collateralAmount);
↪  // this is the real amount


    params.numbers.triggerPrice = triggerPrice;
    params.numbers.acceptablePrice = acceptablePrice; // I can buy with this
↪  price or lower effective spread control
    params.numbers.executionFee = 3200;
```

```
    params.numbers.callbackGasLimit = 3200;
    params.numbers.minOutputAmount = 100; // use the control the final
↪   collateral amount, not for the position size delta, which is indirectly
↪   controlled by acceptable price

    params.orderType = Order.OrderType.LimitDecrease;
    params.decreasePositionSwapType = Order.DecreasePositionSwapType.NoSwap;
    params.isLong = isLong;
    params.shouldUnwrapNativeToken = false;
    params.referralCode = keccak256(abi.encode("MY REFERRAL"));

    vm.prank(account);
    bytes32 key = _erouter.createOrder(params);
    return key;
}


function createLimitIncreaseOrder(address account, address marketToken, address
↪   collateralToken, uint256 collateralAmount, uint sizeDeltaUsd, uint
↪   triggerPrice, bool isLong) public returns(bytes32)
{
    address[] memory swapPath;

    //address[] memory swapPath = new address[](3);
    //swapPath[0] = _marketPropsAB.marketToken;
    //swapPath[1] = _marketPropsBC.marketToken;
    //swapPath[2] = _marketPropsCwnt.marketToken;


    vm.prank(account);
    _wnt.transfer(address(_orderVault), 3200);  // execution fee


    BaseOrderUtils.CreateOrderParams memory params;
    params.addresses.receiver = account;
    params.addresses.callbackContract = address(0);
    params.addresses.uiFeeReceiver = uiFeeReceiver;
    params.addresses.market = marketToken; // final market
    params.addresses.initialCollateralToken = collateralToken; // initial token
    params.addresses.swapPath = swapPath;

    params.numbers.sizeDeltaUsd = sizeDeltaUsd;
    // params.numbers.initialCollateralDeltaAmount = ; // this is actually
↪   useless, will be overidden by real transfer amount
     vm.prank(account);
    IERC20(collateralToken).transfer(address(_orderVault), collateralAmount);
↪   // this is the real amount
```

```
    params.numbers.triggerPrice = triggerPrice;   // used for limit order
    params.numbers.acceptablePrice = 121000000000000; // I can buy with this
↪   price or lower effective spread control
    params.numbers.executionFee = 3200;
    params.numbers.callbackGasLimit = 3200;
    params.numbers.minOutputAmount = 100; // use the control the final
↪   collateral amount, not for the position size delta, which is indirectly
↪   controlled by acceptable price

    params.orderType = Order.OrderType.LimitIncrease;
    params.decreasePositionSwapType = Order.DecreasePositionSwapType.NoSwap;
    params.isLong = isLong;
    params.shouldUnwrapNativeToken = false;
    params.referralCode = keccak256(abi.encode("MY REFERRAL"));

    vm.prank(account);
    bytes32 key = _erouter.createOrder(params);
    return key;
}




function createMarketDecreaseOrder(address account, address marketToken, address
↪   collateralToken,  uint256 acceptablePrice, uint256 sizeInUsd, bool isLong)
↪   public returns(bytes32)
{
    address[] memory swapPath;

    //address[] memory swapPath = new address[](3);
    //swapPath[0] = _marketPropsAB.marketToken;
    //swapPath[1] = _marketPropsBC.marketToken;
    //swapPath[2] = _marketPropsCwnt.marketToken;


    vm.prank(account);
    _wnt.transfer(address(_orderVault), 3200);  // execution fee


    BaseOrderUtils.CreateOrderParams memory params;
    params.addresses.receiver = account;
    params.addresses.callbackContract = address(0);
    params.addresses.uiFeeReceiver = uiFeeReceiver;
    params.addresses.market = marketToken; // final market
    params.addresses.initialCollateralToken = collateralToken; // initial token
    params.addresses.swapPath = swapPath;
```

```
    params.numbers.sizeDeltaUsd = sizeInUsd; // how much dollar to decrease,
↪   will convert into amt of tokens to decrease in long/short based on the
↪   execution price
    params.numbers.initialCollateralDeltaAmount = 13e18; // this is actually
↪   useless, will be overidden by real transfer amount
    // vm.prank(account);
    // IERC20(collateralToken).transfer(address(_orderVault), collateralAmount);
↪    // this is the real amount


    params.numbers.triggerPrice = 0;
    params.numbers.acceptablePrice = acceptablePrice; // I can buy with this
↪   price or lower effective spread control
    params.numbers.executionFee = 3200;
    params.numbers.callbackGasLimit = 3200;
    params.numbers.minOutputAmount = 10e18; // use the control the final
↪   collateral amount, not for the position size delta, which is indirectly
↪   controlled by acceptable price

    params.orderType = Order.OrderType.MarketDecrease;
    params.decreasePositionSwapType = Order.DecreasePositionSwapType.NoSwap;
    params.isLong = isLong;
    params.shouldUnwrapNativeToken = false;
    params.referralCode = keccak256(abi.encode("MY REFERRAL"));

    vm.prank(account);
    bytes32 key = _erouter.createOrder(params);
    return key;
}



function createMarketIncreaseOrder(address account, address marketToken, address
↪   collateralToken, uint256 collateralAmount, uint sizeDeltaUsd, uint
↪   acceptablePrice, bool isLong) public returns(bytes32)
{
    address[] memory swapPath;

    //address[] memory swapPath = new address[](3);
    //swapPath[0] = _marketPropsAB.marketToken;
    //swapPath[1] = _marketPropsBC.marketToken;
    //swapPath[2] = _marketPropsCwnt.marketToken;


    vm.prank(account);
    _wnt.transfer(address(_orderVault), 3200);  // execution fee
```

```
    BaseOrderUtils.CreateOrderParams memory params;
    params.addresses.receiver = account;
    params.addresses.callbackContract = address(0);
    params.addresses.uiFeeReceiver = uiFeeReceiver;
    params.addresses.market = marketToken; // final market
    params.addresses.initialCollateralToken = collateralToken; // initial token
    params.addresses.swapPath = swapPath;

    params.numbers.sizeDeltaUsd = sizeDeltaUsd;
    // params.numbers.initialCollateralDeltaAmount = ; // this is actually
↪   useless, will be overidden by real transfer amount
     vm.prank(account);
    IERC20(collateralToken).transfer(address(_orderVault), collateralAmount);
↪   // this is the real amount


    params.numbers.triggerPrice = 0;
    params.numbers.acceptablePrice = acceptablePrice; // I can buy with this
↪   price or lower effective spread control
    params.numbers.executionFee = 3200;
    params.numbers.callbackGasLimit = 3200;
    params.numbers.minOutputAmount = 100; // use the control the final
↪   collateral amount, not for the position size delta, which is indirectly
↪   controlled by acceptable price

    params.orderType = Order.OrderType.MarketIncrease;
    params.decreasePositionSwapType = Order.DecreasePositionSwapType.NoSwap;
    params.isLong = isLong;
    params.shouldUnwrapNativeToken = false;
    params.referralCode = keccak256(abi.encode("MY REFERRAL"));

    vm.prank(account);
    bytes32 key = _erouter.createOrder(params);
    return key;
}



function createWithdraw(address withdrawor, uint marketTokenAmount) public
↪   returns (bytes32)
{
    address[] memory longTokenSwapPath;
    address[] memory shortTokenSwapPath;

    console.log("createWithdraw with withdrawor: ");
    console.logAddress(withdrawor);
     vm.prank(withdrawor);
    _wnt.transfer(address(_withdrawalVault), 3200);  // execution fee
```

```
    vm.prank(withdrawor);
    ERC20(_marketProps1.marketToken).transfer(address(_withdrawalVault),
↪   marketTokenAmount);

    WithdrawalUtils.CreateWithdrawalParams memory params =
↪   WithdrawalUtils.CreateWithdrawalParams(
        withdrawor, // receiver
        address(0), // call back function
        uiFeeReceiver, // uiFeeReceiver
        _marketProps1.marketToken, // which market token to withdraw
        longTokenSwapPath,
        shortTokenSwapPath,
        123, // minLongTokenAmount
        134, // minShortTokenAmount
        false, // shouldUnwrapNativeToken
        3200, // execution fee
        3200 // callback gas limit
    );

    vm.prank(withdrawor);
    bytes32 key =   _erouter.createWithdrawal(params);
    return key;
}


function createDepositNoSwap(Market.Props memory marketProps, address depositor,
↪   uint amount, bool isLong) public returns (bytes32){
    address[] memory longTokenSwapPath;
    address[] memory shortTokenSwapPath;

    console.log("createDeposit with depositor: ");
    console.logAddress(depositor);

     vm.prank(depositor);
    _wnt.transfer(address(_depositVault), 3200);  // execution fee
    if(isLong){
            console2.log("000000000000000000");
           vm.prank(depositor);
           IERC20(marketProps.longToken).transfer(address(_depositVault),
↪   amount);
           console2.log("bbbbbbbbbbbbbbbbbbbbb");
    }
    else   {
        console2.log("111111111111111111111");
        console2.log("deposit balance: %d, %d",
↪   IERC20(marketProps.shortToken).balanceOf(depositor), amount);
           vm.prank(depositor);
```

```
            IERC20(marketProps.shortToken).transfer(address(_depositVault),
↪   amount);
            console2.log("qqqqqqqqqqqqqqqqqq");
    }


    DepositUtils.CreateDepositParams memory params =
↪   DepositUtils.CreateDepositParams(
        depositor,
        address(0),
        uiFeeReceiver,
        marketProps.marketToken,
        marketProps.longToken,
        marketProps.shortToken,
        longTokenSwapPath,
        shortTokenSwapPath,
        100000, // minMarketTokens
        true,
        3200,  // execution fee
        3200     // call back gas limit
    );

    console2.log("aaaaaaaaaaaaaaaaaaaaaaaaa");
    vm.prank(depositor);
    bytes32 key1 = _erouter.createDeposit(params);

    return key1;
}

/*
function testCancelDeposit() public
{
    address[] memory longTokenSwapPath;
    address[] memory shortTokenSwapPath;

    address(_wnt).call{value: 100e8}("");
    _wnt.transfer(address(_depositVault), 1e6);
    DepositUtils.CreateDepositParams memory params =
↪   DepositUtils.CreateDepositParams(
        msg.sender,
        address(0),
        address(111),
        _marketProps1.marketToken,
        _marketProps1.longToken,
        _marketProps1.shortToken,
        longTokenSwapPath,
        shortTokenSwapPath,
        100000, // minMarketTokens
```

```
        true,
        3200,  // execution fee
        3200     // call back gas limit
    );

    bytes32 key1 = _erouter.createDeposit(params);

    console.log("WNT balance of address(222) before cancelllation: %s",
↪   _wnt.balanceOf(address(222)));
    console.log("WNT balance of address(this) before cancelllation: %s",
↪   _wnt.balanceOf(address(this)));

    _roleStore.grantRole(address(222), Role.CONTROLLER); // to save a market's
↪   props
    vm.prank(address(222));
     _depositHandler.cancelDeposit(key1);
    console.log("WNT balance of address(222) after cancelllation: %s",
↪   _wnt.balanceOf(address(222)));
    console.log("WNT balance of address(this) after cancelllation: %s",
↪   _wnt.balanceOf(address(this)));
}
*/

function testERC165() public{
    bool yes = _wnt.supportsInterface(type(IWNT).interfaceId);
    console2.log("wnt suppports deposit?");
    console2.logBool(yes);
    vm.expectRevert();
    yes = IERC165(address(_sol)).supportsInterface(type(IWNT).interfaceId);
    console2.logBool(yes);

    if(ERC165Checker.supportsERC165(address(_wnt))){
        console2.log("_wnt supports ERC165");
    }
    if(ERC165Checker.supportsERC165(address(_sol))){
        console2.log("_sol supports ERC165");
    }
}

    function justError() external {
        // revert Unauthorized("abcdefg"); // 973d02cb
        // revert("abcdefg");  // 0x08c379a, Error selector
        // require(false, "abcdefg"); // 0x08ce79a, Error selector
        assert(3 == 4); // Panic: 0x4e487b71
    }

     function testErrorMessage() public{
```

```
        try this.justError(){}
        catch (bytes memory reasonBytes) {
            (string memory msg, bool ok ) =
↪   ErrorUtils.getRevertMessage(reasonBytes);
            console2.log("Error Message: "); console2.logString(msg);
            console2.log("error?"); console2.logBool(ok);
        }
    }


    function printAddresses() public{
        console2.log("_orderVault:"); console2.logAddress(address(_orderVault));
        console2.log("marketToken:");
↪   console2.logAddress(address(_marketProps1.marketToken));
    }

    function printPoolsAmounts() public{
        console2.log("\n The summary of pool amounts: ");

        uint256 amount = MarketUtils.getPoolAmount(_dataStore, _marketProps1,
↪   _marketProps1.longToken);
        console2.log("Market: _marketProps1, token: long/nwt, amount: %d",
↪   amount);
        amount = MarketUtils.getPoolAmount(_dataStore, _marketProps1,
↪   _marketProps1.shortToken);
        console2.log("Market: _marketProps1, token: short/USDC, amount: %d",
↪   amount);

        amount = MarketUtils.getPoolAmount(_dataStore, _marketPropsAB,
↪   _marketPropsAB.longToken);
        console2.log("Market: _marketPropsAB, token: long/A, amount: %d",
↪   amount);
        amount = MarketUtils.getPoolAmount(_dataStore, _marketPropsAB,
↪   _marketPropsAB.shortToken);
        console2.log("Market: _marketPropsAB, token: short/B, amount: %d",
↪   amount);

        amount = MarketUtils.getPoolAmount(_dataStore, _marketPropsBC,
↪   _marketPropsBC.longToken);
        console2.log("Market: _marketPropsBC, token: long/B, amount:%d", amount);
        amount = MarketUtils.getPoolAmount(_dataStore, _marketPropsBC,
↪   _marketPropsBC.shortToken);
        console2.log("Market: _marketPropsBC, token: short/C, amount: %d",
↪   amount);

        amount = MarketUtils.getPoolAmount(_dataStore, _marketPropsCwnt,
↪   _marketPropsCwnt.longToken);
```

```
        console2.log("Market: _marketPropsCwnt, token: long/C, amount: %d",
↪   amount);
        amount = MarketUtils.getPoolAmount(_dataStore, _marketPropsCwnt,
↪   _marketPropsCwnt.shortToken);
        console2.log("Market: _marketPropsCwnt, token: short/wnt, amount: %d",
↪   amount);


        console2.log("\n");
    }

}
```

Impact

PositionUtils.validatePosition() uses `isIncrease` instead of `false` when calling isPositionLiquidatable(), making it not work properly for the case of `isIncrease = true`. A liquidation should always be considered as a decrease order in terms of evaluating price impact.

Code Snippet

Tool used

VSCode

Manual Review

Recommendation

Pass false always to isPositionLiquidatable():

```
function validatePosition(
        DataStore dataStore,
        IReferralStorage referralStorage,
        Position.Props memory position,
        Market.Props memory market,
        MarketUtils.MarketPrices memory prices,
        bool isIncrease,
        bool shouldValidateMinPositionSize,
        bool shouldValidateMinCollateralUsd
    ) public view {
        if (position.sizeInUsd() == 0 || position.sizeInTokens() == 0) {
            revert Errors.InvalidPositionSizeValues(position.sizeInUsd(),
↪   position.sizeInTokens());
        }
```

```
        MarketUtils.validateEnabledMarket(dataStore, market.marketToken);
        MarketUtils.validateMarketCollateralToken(market,
↪   position.collateralToken());

        if (shouldValidateMinPositionSize) {
            uint256 minPositionSizeUsd =
↪   dataStore.getUint(Keys.MIN_POSITION_SIZE_USD);
            if (position.sizeInUsd() < minPositionSizeUsd) {
                revert Errors.MinPositionSize(position.sizeInUsd(),
↪   minPositionSizeUsd);
            }
        }

        if (isPositionLiquidatable(
            dataStore,
            referralStorage,
            position,
            market,
            prices,
-            isIncrease,
+           false,
            shouldValidateMinCollateralUsd
        )) {
            revert Errors.LiquidatablePosition();
        }
    }
```

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/pull/155/commits/ac2b1dc0fce1fc73d65859c95eb0ce72d20ff30d

the code was updated to check for whether a position is liquidatable after state variables were updated

Issue M-4: short side of getReservedUsd does not work for market that has the same collateral token

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/198

Found by

0xGoodess

Summary

short side of getReservedUsd does not work for market that has the same collateral token

Vulnerability Detail

Consider the case of ETH / USD market with both long and short collateral token as ETH.

the available amount to be reserved (ETH) would CHANGE with the price of ETH.

```
function getReservedUsd(
    DataStore dataStore,
    Market.Props memory market,
    MarketPrices memory prices,
    bool isLong
) internal view returns (uint256) {
    uint256 reservedUsd;
    if (isLong) {
        // for longs calculate the reserved USD based on the open interest and
↪   current indexTokenPrice
        // this works well for e.g. an ETH / USD market with long collateral
↪   token as WETH
        // the available amount to be reserved would scale with the price of ETH
        // this also works for e.g. a SOL / USD market with long collateral
↪   token as WETH
        // if the price of SOL increases more than the price of ETH, additional
↪   amounts would be
        // automatically reserved
        uint256 openInterestInTokens = getOpenInterestInTokens(dataStore,
↪   market, isLong);
        reservedUsd = openInterestInTokens * prices.indexTokenPrice.max;
    } else {
        // for shorts use the open interest as the reserved USD value
        // this works well for e.g. an ETH / USD market with short collateral
↪   token as USDC
```

```
        // the available amount to be reserved would not change with the price
↪   of ETH
        reservedUsd = getOpenInterest(dataStore, market, isLong);
    }

    return reservedUsd;
}
```

Impact

reservedUsd does not work when long and short collateral tokens are the same.

Code Snippet

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L1415-L1439

Tool used

Manual Review

Recommendation

Consider apply both long and short calculations of reserveUsd with relation to the indexTokenPrice.

Discussion

IIIIIII000

leaning towards invalid, but will let sponsor verify

xvi10

it would not be advisable to use non-stablecoins to back short positions

in case a non-stablecoin is used to back short positions, the amount to be reserved may not need to be changed since the reserve ratio should still validate if the total open interest is a reasonable ratio of the pool's USD value

IIIIIII000

@xvi10 inadvisable, and the stipulation that open interest be a 'reasonable ratio' seem to indication that this bug is possible, and should therefore remain open - am I misunderstanding what you've said

xvi10

the issue does not seem valid to me and the current contract code seems reasonable, an example:

1. there is $50m worth of ETH in the pool and the reserve factor is set to 0.25
2. a max of $12.5m shorts can be opened
3. when validating the reserve we check that the max short open interest does not exceed this
4. if the price of ETH decreases and is now worth $40m, the validation would be that the max open interest does not exceed $10m

the pending pnl of the short positions would increase if the price of ETH decreases, which is a problem with choosing to use a non-stablecoin to back short positions rather than an issue with the validation

in that case the cap of trader pnl and ADL could help to reduce the risk of the market becoming insolvent, but it would be better to avoid the situation by not using a non-stablecoin to back short positions

IIIIIII000

> which is a problem with choosing to use a non-stablecoin to back short positions rather than an issue with the validation

I can see that argument, but you seem to be indicating that users are allowed to do it anyway, and markets becoming insolvent seems like a situation that should be prevented. I'll let Sherlock decide

xvi10

markets have a risk of becoming insolvent, capping trader pnl and ADL helps, reducing the risk of insolvency is left up to the market creator to configure the backing tokens and parameters

hrishibhat

Considering this issue as a valid medium based on the above comments that there is a possibility of market insolvency

Issue M-5: Keepers can steal additional execution fee from users

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/199

Found by

J4de, KingNFT

Summary

The implementation of `payExecutionFee()` didn't take EIP-150 into consideration, a malicious keeper can exploit it to drain out all execution fee users have paid, regardless of the actual execution cost.

Vulnerability Detail

The issue arises on `L55` of `payExecutionFee()`, as it's an `external` function, calling`payExecutionFee()` is subject to EIP-150. Only `63/64` gas is passed to the `GasUtils` sub-contract(`external library`), and the remaing `1/64` gas is reserved in the caller contract which will be refunded to keeper(`msg.sender`) after the execution of the whole transaction. But calculation of `gasUsed` includes this portion of the cost as well.

```
File: contracts\gas\GasUtils.sol
46:     function payExecutionFee(
47:         DataStore dataStore,
48:         EventEmitter eventEmitter,
49:         StrictBank bank,
50:         uint256 executionFee,
51:         uint256 startingGas,
52:         address keeper,
53:         address user
54:     ) external { // @audit external call is subject to EIP-150
-55:         uint256 gasUsed = startingGas - gasleft();
+           uint256 gasUsed = startingGas - gasleft() * 64 / 63; // @audit the
 ↪  correct formula
56:         uint256 executionFeeForKeeper = adjustGasUsage(dataStore, gasUsed) *
 ↪  tx.gasprice;
57:
58:         if (executionFeeForKeeper > executionFee) {
59:             executionFeeForKeeper = executionFee;
60:         }
61:
62:         bank.transferOutNativeToken(
63:             keeper,
```

```
64:             executionFeeForKeeper
65:         );
66:
67:         emitKeeperExecutionFee(eventEmitter, keeper, executionFeeForKeeper);
68:
69:         uint256 refundFeeAmount = executionFee - executionFeeForKeeper;
70:         if (refundFeeAmount == 0) {
71:             return;
72:         }
73:
74:         bank.transferOutNativeToken(
75:             user,
76:             refundFeeAmount
77:         );
78:
79:         emitExecutionFeeRefund(eventEmitter, user, refundFeeAmount);
80:     }
```

A malicious keeper can exploit this issue to drain out all execution fee, regardless of the actual execution cost. Let's take `executeDeposit()` operation as an example to show how it works:

```
File: contracts\exchange\DepositHandler.sol
092:     function executeDeposit(
093:         bytes32 key,
094:         OracleUtils.SetPricesParams calldata oracleParams
095:     ) external
096:         globalNonReentrant
097:         onlyOrderKeeper
098:         withOraclePrices(oracle, dataStore, eventEmitter, oracleParams)
099:     {
100:         uint256 startingGas = gasleft();
101:
102:         try this._executeDeposit(
103:             key,
104:             oracleParams,
105:             msg.sender
106:         ) {
107:         } catch (bytes memory reasonBytes) {
...
113:         }
114:     }

File: contracts\exchange\DepositHandler.sol
141:     function _executeDeposit(
142:         bytes32 key,
143:         OracleUtils.SetPricesParams memory oracleParams,
```

```
144:         address keeper
145:     ) external onlySelf {
146:         uint256 startingGas = gasleft();
...
171:
172:         ExecuteDepositUtils.executeDeposit(params);
173:     }


File: contracts\deposit\ExecuteDepositUtils.sol
096:     function executeDeposit(ExecuteDepositParams memory params) external {
...
221:
222:         GasUtils.payExecutionFee(
223:             params.dataStore,
224:             params.eventEmitter,
225:             params.depositVault,
226:             deposit.executionFee(),
227:             params.startingGas,
228:             params.keeper,
229:             deposit.account()
230:         );
231:     }

File: contracts\gas\GasUtils.sol
46:     function payExecutionFee(
47:         DataStore dataStore,
48:         EventEmitter eventEmitter,
49:         StrictBank bank,
50:         uint256 executionFee,
51:         uint256 startingGas,
52:         address keeper,
53:         address user
54:     ) external {
55:         uint256 gasUsed = startingGas - gasleft();
56:         uint256 executionFeeForKeeper = adjustGasUsage(dataStore, gasUsed) *
↪  tx.gasprice;
57:
58:         if (executionFeeForKeeper > executionFee) {
59:             executionFeeForKeeper = executionFee;
60:         }
61:
62:         bank.transferOutNativeToken(
63:             keeper,
64:             executionFeeForKeeper
65:         );
66:
67:         emitKeeperExecutionFee(eventEmitter, keeper, executionFeeForKeeper);
```

```
68:
69:         uint256 refundFeeAmount = executionFee - executionFeeForKeeper;
70:         if (refundFeeAmount == 0) {
71:             return;
72:         }
73:
74:         bank.transferOutNativeToken(
75:             user,
76:             refundFeeAmount
77:         );
78:
79:         emitExecutionFeeRefund(eventEmitter, user, refundFeeAmount);
80:     }

File: contracts\gas\GasUtils.sol
097:     function adjustGasUsage(DataStore dataStore, uint256 gasUsed) internal
↪  view returns (uint256) {
...
105:         uint256 baseGasLimit =
↪  dataStore.getUint(Keys.EXECUTION_GAS_FEE_BASE_AMOUNT);
...
109:         uint256 multiplierFactor =
↪  dataStore.getUint(Keys.EXECUTION_GAS_FEE_MULTIPLIER_FACTOR);
110:         uint256 gasLimit = baseGasLimit + Precision.applyFactor(gasUsed,
↪  multiplierFactor);
111:         return gasLimit;
112:     }
```

To simplify the problem, given

```
EXECUTION_GAS_FEE_BASE_AMOUNT = 0
EXECUTION_GAS_FEE_MULTIPLIER_FACTOR = 1
executionFeeUserHasPaid = 200K Gwei
tx.gasprice = 1 Gwei
actualUsedGas = 100K
```

`actualUsedGas` is the gas cost since `startingGas`(L146 of `DepositHandler.sol`) but before calling `payExecutionFee()`(L221 of `ExecuteDepositUtils.sol`)

Let's say, the keeper sets `tx.gaslimit` to make

```
startingGas = 164K
```

Then the calculation of `gasUsed`, L55 of `GasUtils.sol`, would be

```
uint256 gasUsed = startingGas - 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 `startingGas = 6500K`, the calculation of `gasUsed` would be

```
uint256 gasUsed = startingGas - 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/2023-04-gmx/blob/main/gmx-synthetics/contracts/gas/GasUtils.sol#L55

Tool used

Manual Review

Recommendation

The description in `Vulnerability Detail` section has been simplified. In fact, `gasleft` value should be adjusted after each external call during the whole call stack, not just in `payExecutionFee()`.

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/pull/155/commits/0e784e36c518dd7c1e0f529f49b733014c50e263

IIIIIIIOOO

https://github.com/gmx-io/gmx-synthetics/commit/0e784e36c518dd7c1e0f529f49b733014c50e263 The commit adjusts the starting amount of gas by the number of external calls being made. Each external call was correctly identified and is adjusted. `gasleft() = startGas - startGas * 63 / 64; start' = gasleft() / 63; start' = (startGas - startGas * 63 / 64) / 63; start' = startGas - (startGas * 1 / 64)` so the math works out. However, because the gas passed along is the floor of the division, if the amount of gas passed to the external call is not an exact multiple of 64, the modulo 64 amount of gas won't be refunded properly. partially reviewed

Issue M-6: An Oracle Signer can never be removed even if he becomes malicious

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/205

Found by

Chinmay

Summary

The call flow of removeOracleSIgner incorrectly compares the hash of ("removeOracleSigner", account) with the hash of ("addOracleSigner", account) for validating that an action is actually initiated. This validation always fails because the hashes can never match.

Vulnerability Detail

The process of removing oracle signers is 2 stage. First function `signalRemoveOracleSigner` is called by the TimelockAdmin which stores a time-delayed timestamp corresponding to the keccak256 hash of ("removeOracleSigner", account) - a bytes32 value called actionKey in the pendingActions mapping.

Then the Admin needs to call function `removeOracleSignerAfterSignal` but this function calls `_addOracleSignerActionKey` instead of `_removeOracleSignerActionKey` for calculating the bytes32 action key value. Now the actionKey is calculated as keccak256 hash of ("addOracleSigner", account) and this hash is used for checking if this action is actually pending by ensuring its timestamp is not zero inside the `_validateAction` function called via `_validateAndClearAction` function at Line 122. The hash of ("removeOracleSigner", account) can never match hash of ("addOracleSigner", account) and thus this validation will fail.

```
function removeOracleSignerAfterSignal(address account) external
↪  onlyTimelockAdmin nonReentrant {
       bytes32 actionKey = _addOracleSignerActionKey(account);
       _validateAndClearAction(actionKey, "removeOracleSigner");

       oracleStore.removeSigner(account);

       EventUtils.EventLogData memory eventData;
       eventData.addressItems.initItems(1);
       eventData.addressItems.setItem(0, "account", account);
       eventEmitter.emitEventLog1(
           "RemoveOracleSigner",
           actionKey,
```

```
            eventData
        );
    }
```

Impact

The process of removing an Oracle Signer will always revert and this breaks an important safety measure if a certain oracle signer becomes malicious the TimelockAdmin could do nothing(these functions are meant for this). Hence, important functionality is permanently broken.

Code Snippet

https://github.com/sherlock-audit/2023-04-gmx/blob/06ccd8c7ee4cadc46e3ac412dcf141eefdced42f/gmx-synthetics/contracts/config/Timelock.sol#L117

Tool used

Manual Review

Recommendation

Replace the call to _addOracleSignerActionKey at Line 118 by call to _removeOracleSignerActionKey

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/pull/155/commits/3ef1161f448941f96c4a7c053391d8b0d63648b6

IIIIIII000

https://github.com/gmx-io/gmx-synthetics/commit/3ef1161f448941f96c4a7c053391d8b0d63648b6 The commit correctly implements the suggested fix of using the correct function (`_removeOracleSignerActionKey()`) for looking up the action key for oracle removal done

Issue M-7: Stop-loss orders do not become marketable orders

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/233

Found by

Ch_301, lllllll, ShadowForce, lemonmon

Summary

Stop-loss orders do not execute if the market has moved since the order was submitted.

Vulnerability Detail

This is the normal behavior for a stop-loss order in financial markets: "A stop-loss order becomes a market order as soon as the stop-loss price is reached. Since the stop loss becomes a market order, execution of the order is guaranteed, but not the price." -

Another way this could happen is if there's an oracle outage, and the oracles come back after the trigger price has been passed.

Impact

Since the stop-loss will revert if a keeper tries to execute it, essentially the order becomes wedged and there will be no stoploss protection.

Consider a scenario where the price of token X is $100, and a user who is long is in a profit above $99, but will have a loss at $98, so they set a stoploss with a trigger price of $99 and submit the order to the mempool. By the time the block gets mined 12 seconds later, the primary and secondary prices are $97/96$, and the order becomes unexecutable.

Code Snippet

Stop-loss orders revert if the primary price and secondary price don't straddle the trigger price

```
// File: gmx-synthetics/contracts/order/BaseOrderUtils.sol : BaseOrderUtils.ok
↪  #1

270
271             if (orderType == Order.OrderType.StopLossDecrease) {
```

```
272                 uint256 primaryPrice =
↪  oracle.getPrimaryPrice(indexToken).pickPrice(shouldUseMaxPrice);
273                 uint256 secondaryPrice =
↪  oracle.getSecondaryPrice(indexToken).pickPrice(shouldUseMaxPrice);
274
275                 // for stop-loss decrease orders:
276                 //     - long: validate descending price
277                 //     - short: validate ascending price
278                 bool shouldValidateAscendingPrice = !isLong;
279
280 @>              bool ok = shouldValidateAscendingPrice ?
281                     (primaryPrice <= triggerPrice && triggerPrice <=
↪  secondaryPrice) :
282                     (primaryPrice >= triggerPrice && triggerPrice >=
↪  secondaryPrice);
283
284                 if (!ok) {
285                     revert Errors.InvalidStopLossOrderPrices(primaryPrice,
↪  secondaryPrice, triggerPrice, shouldValidateAscendingPrice);
286:                }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/order/BaseOrderUtils.sol#L270-L286

The revert is allowed to cause the order to fail:

```
// File: gmx-synthetics/contracts/exchange/OrderHandler.sol :
↪  OrderHandler._handleOrderError()   #2

263                 // the transaction is reverted for InvalidLimitOrderPrices and
264                 // InvalidStopLossOrderPrices errors since since the oracle
↪  prices
265                 // do not fulfill the specified trigger price
266                 errorSelector == Errors.InvalidLimitOrderPrices.selector ||
267 @>              errorSelector == Errors.InvalidStopLossOrderPrices.selector
268             ) {
269                 ErrorUtils.revertWithCustomError(reasonBytes);
270:            }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/exchange/OrderHandler.sol#L263-L270

Tool used

Manual Review

Recommendation

Don't revert if both the primary and secondary prices are worse than the trigger price.

Discussion

xvi10

would classify this as a medium, the oracle should sign a price in the block in which the order was submitted, the likelihood that the price was out of range by then should be small, it is somewhat similar to the chance of submitting a market order and price moves past the acceptable price

Jiaren-tang

Escalate for 10 USDC.

the severity should be high.

the impact:

> Since the stop-loss will revert if a keeper tries to execute it, essentially the order becomes wedged and there will be no stoploss protection

basically this is equal to lack of slippage protection!

sherlock-admin

> Escalate for 10 USDC.
>
> the severity should be high.
>
> the impact:
>
> > Since the stop-loss will revert if a keeper tries to execute it, essentially the order becomes wedged and there will be no stoploss protection
>
> basically this is equal to lack of slippage protection!

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

lllllll000

Will leave this up to Sherlock

hrishibhat

Result: Medium Has duplicates In addition to the Sponsor comment, This requires the market to change before the order gets mined or an oracle outage. This is a valid medium.

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- ShadowForce: rejected

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/pull/155/commits/3243138ebdc6f86e06f5b1fef91312113ef36e20

the code was updated to allow stop-loss orders to be executed if the trigger price was crossed

Issue M-8: Users can get impact pool discounts while also increasing the virtual impact pool skew

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/246

Found by

lllllll

Summary

Virtual impacts will remain consistently negative, even if most pools' impacts are balanced

Vulnerability Detail

`applyDeltaToPoolAmount()`/`applyDeltaToVirtualInventoryForPositions()` always modify the virtual impact, even if the virtual impact wasn't consulted

Impact

Imagine a different stablecoin is one of the tokens in each of 5 markets where the other side is Eth, all five markes are part of the same virtual market, and each of the markets' swap imbalance is such that users will get a discount for swapping from Eth into the stablecoin, and the global virtual swap balance for 'stable'/Eth is flat. Users will keep swapping from Eth into the stable, which will push the individual pool towards being flat, but will push the virtual swap inventory for stables to be more and more negative. As soon as one of the five markets becomes flat, any user trying to swap Eth into a stable will suddenly be hit with the huge virtual swap impact imbalance

Code Snippet

`applyDeltaToPoolAmount()` unconditionally updates the virtual inventory for swaps:

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol :
↪  MarketUtils.applyDeltaToPoolAmount()   #1

711 @>          applyDeltaToVirtualInventoryForSwaps(
712                 dataStore,
713                 eventEmitter,
714                 market,
715                 token,
```

```
716                 delta
717:            );
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L711-L717

but the actual price impact skips the virtual impact if one of the tokens doesn't have virtual inventory:

```
// File: gmx-synthetics/contracts/pricing/SwapPricingUtils.sol :
↪  SwapPricingUtils.getPriceImpactUsd()   #2

119 @>          if (!hasVirtualInventoryTokenA || !hasVirtualInventoryTokenB) {
120                 return priceImpactUsd;
121:            }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/pricing/SwapPricingUtils.sol#L119-L121

or if the impact was positive:

```
// File: gmx-synthetics/contracts/pricing/SwapPricingUtils.sol :
↪  SwapPricingUtils.getPriceImpactUsd()   #3

96              // the virtual price impact calculation is skipped if the price
↪  impact
97              // is positive since the action is helping to balance the pool
98              //
99              // in case two virtual pools are unbalanced in a different
↪  direction
100             // e.g. pool0 has more WNT than USDC while pool1 has less WNT
101             // than USDT
102             // not skipping the virtual price impact calculation would lead to
103             // a negative price impact for any trade on either pools and would
104             // disincentivise the balancing of pools
105:@>          if (priceImpactUsd >= 0) { return priceImpactUsd; }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/pricing/SwapPricingUtils.sol#L91-L104

Virtual position impacts have the same issue

Tool used

Manual Review

Recommendation

Don't adjust the virtual inventory for swaps if the virtual inventory wasn't consulted when calculating the impact.

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/pull/155/commits/1b35173e915a7d3bb85f6c6b6e59a7210897854c

virtual swap price impact should be tracked by virtualInventoryForSwapsKey(virtualMarketId, isLongToken) instead of virtualInventoryForSwapsKey(virtualMarketId, token)

this would allow similar markets e.g. ETH/USDC, ETH/USDT to be associated together

Issue M-9: Virtual swap balances don't take into account token prices

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/251

Found by

lllllll

Summary

Virtual swap balances don't take into the fact that an exchange between the collateral token and the virtual token is taking place, necessitating an exchange rate.

Vulnerability Detail

Virtual position impacts are all based on the virtual token being the same token as the market's index token. With virtual swap impacts, they're not the same token - the market token is a market collateral token, and the virtual token is some other token, which likely has a different price. For example, the README mentions ETH/USDC as ETH/USDT, where USDC is paired with USDT. USDC and USDT sound like they should be equivalent, but looking at the monthly chart of the exchange rate between the two, it has been between 0.8601 and 1.2000 - a 20% difference at the margins. Further, if one of them were to de-peg, the difference may be even larger and for a much longer period of time.

This applies to swaps, as well as to position changes, since those also track virtual swap inventory, since the balance is changing.

Impact

Orders on some markets will get larger/smaller virtual discounts/penalties than they should, as compared to other markets using the same virtual impact pool. In addition to the basic accounting/fee issues associated with the difference, if the price difference is large enough, someone can swap through the market where the impact is smaller due to the exchange rate in order to push the impact more negative, and then simultaneously swap through the other market, where the same amount of funds would result in a larger positive impact than was incurred negatively in the other market, unfairly draining any impact discounts available to legitimate traders.

Code Snippet

The delta being applied is a token amount:

```
// File: gmx-synthetics/contracts/swap/SwapUtils.sol : SwapUtils._swap()   #1

283             MarketUtils.applyDeltaToPoolAmount(
284                 params.dataStore,
285                 params.eventEmitter,
286                 _params.market.marketToken,
287                 _params.tokenIn,
288 @>              (cache.amountIn + fees.feeAmountForPool).toInt256()
289             );
290
291             // the poolAmountOut excludes the positive price impact amount
292             // as that is deducted from the swap impact pool instead
293:            MarketUtils.applyDeltaToPoolAmount(
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/swap/SwapUtils.sol#L273-L293

and is stored unaltered as the virtual amount:

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol :
↪  MarketUtils.applyDeltaToVirtualInventoryForSwaps()   #2

1471        function applyDeltaToVirtualInventoryForSwaps(
1472            DataStore dataStore,
1473            EventEmitter eventEmitter,
1474            address market,
1475            address token,
1476            int256 delta
1477        ) internal returns (bool, uint256) {
1478            bytes32 marketId =
↪  dataStore.getBytes32(Keys.virtualMarketIdKey(market));
1479            if (marketId == bytes32(0)) {
1480                return (false, 0);
1481            }
1482
1483            uint256 nextValue = dataStore.applyBoundedDeltaToUint(
1484                Keys.virtualInventoryForSwapsKey(marketId, token),
1485 @>             delta
1486            );
1487
1488            MarketEventUtils.emitVirtualSwapInventoryUpdated(eventEmitter,
↪  market, token, marketId, delta, nextValue);
1489
1490            return (true, nextValue);
1491:       }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L1471-L1491

The same is true for deposits, decreases, increases, and withdrawals.

Tool used

Manual Review

Recommendation

Use oracle prices and convert the collateral token to the specific virtual token

Discussion

xvi10

added a note in https://github.com/gmx-io/gmx-synthetics/commit/3f17dd59b482e202b52652f8191581ca3827b18e

IIIIIII000

https://github.com/gmx-io/gmx-synthetics/commit/3f17dd59b482e202b52652f8191581ca3827b18e The commit does not fix the issue, and instead acknowledges it via code comments, explaining the issue and its potential negative effects, and says that a patch may be applied at a later time if the issue ever manifests and needs to be addressed. done

Issue M-10: Virtual swap impacts can be bypassed by swapping through markets where only one of the collateral tokens has virtual inventory

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/257

Found by

lllllll

Summary

Virtual swap impacts can be bypassed by swapping through markets where only one of the collateral tokens has virtual inventory

Vulnerability Detail

The code that calculates price impacts related to swapping, skips the application of virtual impacts if one of the tokens doesn't have a virtual token set

Impact

If the virtual swap amount for a particular token is very large, and a large swap through that market would cause the balance to drop a lot, causing the trade to have a large negative impact, a user can split their large order into multiple smaller orders, and route them through other markets where there is no virtual token for one of the pools, and avoid the fees (assuming those pools have non-virtual imbalances that favor such a trade).

Code Snippet

Virtual impacts are completely skipped if one of the tokens doesn't have a virtual version:

```
// File: gmx-synthetics/contracts/pricing/SwapPricingUtils.sol :
↪  SwapPricingUtils.getPriceImpactUsd()   #1

113             (bool hasVirtualInventoryTokenB, uint256
↪  virtualPoolAmountForTokenB) = MarketUtils.getVirtualInventoryForSwaps(
114                 params.dataStore,
115                 params.market.marketToken,
116                 params.tokenB
117             );
118
119             if (!hasVirtualInventoryTokenA || !hasVirtualInventoryTokenB) {
```

```
120 @>              return priceImpactUsd;
121:            }
```

https://github.com/sherlock-audit/2023-04-gmx/blob/main/gmx-synthetics/contracts/pricing/SwapPricingUtils.sol#L113-L121

Tool used

Manual Review

Recommendation

Use the non-virtual token's inventory as the standin for the missing virtual inventory token

Discussion

xvi10

fixed in https://github.com/gmx-io/gmx-synthetics/pull/155/commits/1b35173e915a7d3bb85f6c6b6e59a7210897854c

virtual swap price impact should be tracked by virtualInventoryForSwapsKey(virtualMarketId, isLongToken) instead of virtualInventoryForSwapsKey(virtualMarketId, token)

this would allow similar markets e.g. ETH/USDC, ETH/USDT to be associated together

Issue M-11: Calc.boundedAdd used intitally but later regular subtraction used

Source: https://github.com/sherlock-audit/2023-04-gmx-judging/issues/269

Found by

stent

Summary

In MarketDecrease order flow the funding amount per size calculation uses `Calc.boundedAdd` but further down the call path there is a calculation that does not use `Calc.boundedSubtract` but rather a regular `-`, which could cause overflow errors.

Vulnerability Detail

In `MarketUtils.getNextFundingAmountPerSize` the funding amount per size variables in the `GetNextFundingAmountPerSizeResult` variable are calculated using `Calc.boundedAdd(a,b)`. If the inputs to the function are `(type(int256).min+10,-12)`, for example, then the value returned will be `type(int256).min` i.e. it will not cause an arithmetic overflow, like it would if the values were just added normally with `+`.

Further down the call path of MarketDecrease in `MarketUtils.getFundingFeeAmount` the funding amount is negated, and negating `type(int256).min` results in an arithmetic overflow.

If the point of defending overflow of the funding amount per size variables was to allow execution to pass without failure then all calculations involving the funding amount per size variables should use `Calc.boundedAdd` or `Calc.boundedSub`.

Impact

MarketDecrease orders that would be expected to pass would fail, causing them to be cancelled.

Code Snippet

```
// contract & func: MarketUtils.getNextFundingAmountPerSize

            result.fundingAmountPerSize_LongCollateral_LongPosition =
↪   Calc.boundedAdd(
                result.fundingAmountPerSize_LongCollateral_LongPosition,
```

```
↪  cache.fps.fundingAmountPerSizeDelta_LongCollateral_LongPosition.toInt256()
        );
```

https://github.com/gmx-io/gmx-synthetics/blob/a2e331f6d0a3b59aaac5ead975b206840369a723/contracts/market/MarketUtils.sol#L1074

```
// contract: Calc

    function boundedAdd(int256 a, int256 b) internal pure returns (int256) {
        // if either a or b is zero or if the signs are different there should
↪  not be any overflows
        if (a == 0 || b == 0 || (a < 0 && b > 0) || (a > 0 && b < 0)) {
            return a + b;
        }

        // if adding `b` to `a` would result in a value less than the min int256
↪  value
        // then return the min int256 value
        if (a < 0 && b <= type(int256).min - a) {
            return type(int256).min;
        }

        // if adding `b` to `a` would result in a value more than the max int256
↪  value
        // then return the max int256 value
        if (a > 0 && b >= type(int256).max - a) {
            return type(int256).max;
        }

        return a + b;
    }
```

https://github.com/gmx-io/gmx-synthetics/blob/a2e331f6d0a3b59aaac5ead975b206840369a723/contracts/utils/Calc.sol#L90

```
// contract: MarketUtils

    function getFundingFeeAmount(
        int256 latestFundingAmountPerSize,
        int256 positionFundingAmountPerSize,
        uint256 positionSizeInUsd
    ) internal pure returns (int256) {
        int256 fundingDiffFactor = (latestFundingAmountPerSize -
↪  positionFundingAmountPerSize);
```

https://github.com/gmx-io/gmx-synthetics/blob/a2e331f6d0a3b59aaac5ead975b

206840369a723/contracts/market/MarketUtils.sol#L1345

Tool used

Manual Review

Recommendation

Use `Calc.boundedAdd` & `Calc.boundedSub` on all calculations involving addition and subtraction with funding amount per size variables.

Discussion

IIIIIIIOOO

will let sponsor review

xvi10

it seems the market could get bricked if reaching this state, despite attempts at avoiding that, the likelihood of this should be quite small

if the market reaches this state, the market may not function correctly, it may be preferable to update the funding fee factors to prevent this state instead of handling this in the contracts, to avoid the complexity of handling this case for this scope

the contracts could be updated in the future to properly handle this case