SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Lyra
Prepared by:	Sherlock
Lead Security Expert:	thec00n
Dates Audited:	December 5 - January 2, 2023
Prepared on:	January 24, 2023

Introduction

The Lyra Protocol is a novel automated market maker that lets you earn trading fees on capital by automatically making options markets.

Scope

```
libraries/**
sythetix/**
contracts/BaseExchangeAdapter.sol
contracts/GMXAdapter.sol
contracts/GMXFuturesPoolHedger.sol
contracts/LiquidityPool.sol
contracts/LiquidityToken.sol
contracts/OptionGreekCache.sol
contracts/OptionMarket.sol
contracts/OptionMarketPricer.sol
contracts/OptionToken.sol
contracts/ShortCollateral.sol
```

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
10	5

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

thec00n	ctf_sec	GalloDaSballo
hyh	Bahurum	TrungOre
clems4ever	hansfriese	Jeiwan

Issue H-1: GMXFuturesPoolHedger's _increasePosition can push the leverage too high and have hedging liquidated

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/79

Found by

hyh

Summary

_increasePosition() add to a position no matter how low amount of collateral was obtained from the LiquidityPool.

If the resulting leverage is too high, the new increased position can be liquidated soon and the pool can find itself with no hedge.

Vulnerability Detail

If `liquidityPool.transferQuoteToHedge(collateralDelta)` result is positive, the position increase will carry on, no matter how much funds were obtained from the LiquidityPool.

I.e. zero amount and dust amount of the funds obtained aren't distinct from the hedging position risk perspective, but are treated differently.

More generally, if ‘collateralDelta > liquidityPool.transferQuoteToHedge(collateralDelta)', the resulting leverage needs to be checked before proceeding as it can be too high.

Impact

As the market changes constantly the liquidation can realistically happen before any manual collateral adjustment is made. Losing the hedging position due to liquidation will mean that the net option position is now unhedged.

Without hedging the protocol is open to any delta originated losses, which can be massive and can have the net impact up to the protocol insolvency, with net position becoming equivalent to the naked option selling.

There is no prerequisites beside lack of collateral that can take place as a part of usual protocol workflow. There is a massive fund loss impact from the loss of hedging position. Given this setting the severity to be high.

Code Snippet

It's now allowed to open a position with any low positive collateral as only `collateralDelta == 0` case is reverted:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L626-L665

```
function _increasePosition(
  PositionDetails memory currentPos,
  bool isLong,
  uint sizeDelta,
  uint collateralDelta,
  uint spot
) internal {
  // add margin fee
  // when we increase position, fee always got deducted from collateral
  collateralDelta += _getPositionFee(currentPos.size, sizeDelta,
↪  currentPos.entryFundingRate);

  address[] memory path;
  uint acceptableSpot;

  if (isLong) {
    path = new address[](2);
    path[0] = address(quoteAsset);
    path[1] = address(baseAsset);
    acceptableSpot = _convertToGMXPrecision(spot.multiplyDecimal(futuresPoolHedg ⌋
↪  erParams.acceptableSpotSlippage));
  } else {
    path = new address[](1);
    path[0] = address(quoteAsset);
    acceptableSpot = _convertToGMXPrecision(spot.divideDecimalRound(futuresPoolH ⌋
↪  edgerParams.acceptableSpotSlippage));
  }

  // if the trade ends up with collateral > size, adjust collateral.
  // gmx restrict position to have size >= collateral, so we cap the collateral
↪  to be same as size.
  if (currentPos.collateral + collateralDelta > currentPos.size + sizeDelta) {
    collateralDelta = (currentPos.size + sizeDelta) - currentPos.collateral;
  }

  // if we get less than we want, we will just continue with the same position,
↪  but take on more leverage
  collateralDelta = liquidityPool.transferQuoteToHedge(collateralDelta);

  if (collateralDelta == 0) {
```

SHERLOCK

```
    revert NoQuoteReceivedFromLP(address(this));
  }

  // collateralDelta with decimals same as defined in ERC20
  collateralDelta = ConvertDecimals.convertFrom18(collateralDelta,
↪   quoteAsset.decimals());
```

transferQuoteToHedge() limits the funds available for PoolHedger by `liquidity.pendingDeltaLiquidity + liquidity.freeLiquidity`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L1037-L1055

```
/**
 * @notice Sends quote to the PoolHedger.
 * @dev Transfer amount up to `pendingLiquidity + freeLiquidity`.
 * The hedger must determine what to do with the amount received.
 *
 * @param amount The amount requested by the PoolHedger.
 */
function transferQuoteToHedge(uint amount) external onlyPoolHedger returns
↪   (uint) {
  Liquidity memory liquidity = getLiquidity();

  uint available = liquidity.pendingDeltaLiquidity + liquidity.freeLiquidity;

  amount = amount > available ? available : amount;

  _transferQuote(address(poolHedger), amount);
  emit QuoteTransferredToPoolHedger(amount);

  return amount;
}
```

Tool used

Manual Review

Recommendation

If the funds available are less than requested there is a choice between under-hedging (not increasing the position) and pushing the leverage too high (increasing it with less collateral than desired).

The preferred option here depends on the resulting leverage. If it is not too high the full position increase is desirable as under-hedging is much more dangerous. But if leverage is being pushed too close to the liquidation threshold, the risk of losing the hedge altogether out-weights the less-then-desired hedging considerations.

This way there is an optimal max leverage parameter that balances these two risks and the decision whether to proceed with that much collateral should be based on the resulting leverage exceeding it or not.

Consider introducing the max leverage parameter or use the global one, say `futuresMarketSettings.maxLeverage(marketKey)`, and reverting the attempts of the hedging position increase that bring the estimated leverage above it.

Such events aren't part of a fully automated workflow and should lead to manual collateral addition and repeating of the hedgeDelta() or updateCollateral() calls.

Discussion

hrishibhat

Sponsor comment:

> Valid, maybe medium - high is okay too

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/82a6f7d9d0ed50e3b8a00a27572138345d1bbe6c

thec00n

`_increasePosition()` checks if the maximum leverage is in acceptable range including the amount from current position update. The function does not attempt to increase to max leverage threshold. Fix LGTM.

Issue H-2: updateCollateral can be used to modify the token price of the liquidity pool

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/19

Found by

clems4ever

Summary

When calling updateCollateral, a request is submitted to Gmx position router, and some quote amount is transferred either in or out of the liquidity pool to the `GMXFuturesPoolHedger`.

In LiquidityPool.sol: `_getTotalPoolValueQuote` uses the variable `usedDeltaLiquidity` to track available quote amounts used on GMX as collateral.

The variable `usedDeltaLiquidity` will not be increased during the period in which the request for increase is pending, but the quote will be sent to GMXFuturesPoolHedger. A malicious user can call `updateCollateral` before depositing into the LP, getting more shares.

Vulnerability Detail

Impact

Code Snippet

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L294

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L869

Tool used

Manual Review

Recommendation

If quote amount has been transferred from liquidityPool, add `pendingDelta` into `totalPoolValue` in `function _getTotalPoolValueQuote`

Discussion

hrishibhat

Sponsor comment:

> Valid This is something we found before the contest but great to see it got spotted Recommendation is not really written correctly, instead the pending update collateral in GMX need to be accounted for in "usedDeltaLiquidity"

thec00n

Fix https://github.com/lyra-finance/lyra-protocol/pull/7/commits/7262e3ea8d4e5bcebf05d4fe16d0917e67e7dd28

thec00n

The amount that is sent from the `LiquidityPool` to the pool hedger when creating an increase position request is fetched from the GMX `PostionRouter` with `_getPendingIncreaseCollateralDelta()` and added to `usedDeltaLiquidity`. Fix LGTM

Issue H-3: `hedgeDelta()` calculates `collateralDelta` inaccurately when there is negative PNL

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/10

Found by

thec00n

Summary

`GMXFuturesPoolHedger` does not calculate the `collateralDelta` correctly when the hedge position has a negative PNL. A price change in either direction that creates a big enough `collateralDelta` could create a scenario where the system can not be hedged anymore, and calls to `hedgeDelta()` will always fail.

Vulnerability Detail

The scenario can be reproduced by setting the price change in the `if spot price decreases, hedger needs to long less` test case to 1200 instead of 1300 in `IntegrationTestsGMX.ts` on line 553. The following output below contains the position when the price is updated and `hedgeDelta()` is called, as well as relevant output from GMX `Vault` when position execution calls `decreasePosition()` and subsequently `_reduceCollateral()` where the call fails at Vault.sol#L1039.

```
--- current position ---
pos.size 14548
pos.collateral 13512
pos.averagePrice 1614
pos.entryFundingRate 0
pos.unrealisedPnl -3738
pos.isLong true
-- Call  Vault.decreasePosition ---
sizeDelta 11847
collateralDelta 10811
--- Reducing position.collateral ---
old position.collateral 13512
adjustedDelta 3044
new position.collateral 10468
--- Adjusting pnl
old pnl -3044
new pnl 0
--- Trying to subtract _collateralDelta from position.collateral ---
position.collateral  10468
_collateralDelta 10811
```

```
!!!FAIL!!!
```

Impact

The system can not hedge its position anymore if the negative PNL is too big in proportion to its size. This could lead to increased financial losses for Lyra LPs.

Code Snippet

https://github.com/sherlock-audit/2022-12-lyra/blob/main/test/contracts/gmx_integration/IntegrationTestsGMX.ts#L548-L583

Tool used

Manual Review

Recommendation

`collateralDelta` needs to account for a negative PNL position and subtract the adjusted delta in the same way that collateral is adjusted downwards in GMX `Vault._reduceCollateral()`.

Discussion

hrishibhat

Sponsor comment:

> Valid

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/6f0ee00187ea91eb92dcfa090fb6c398b8b51822

thec00n

`collateralDelta` is reduced proportionally in case the current PNL is negative (see https://github.com/lyra-finance/lyra-protocol/blob/997a65f8f7f24071882e71254a1863c19626e341/contracts/GMXFuturesPoolHedger.sol#L794). LGTM.

Issue H-4: `processWithdrawalQueue` can permanently fail due to blacklisted addresses

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/6

Found by

thec00n

Summary

Withdrawals are made in a two-step process. First, a user starts the withdrawal, and if the initial validation succeeds, the withdrawal is added to `queuedWithdrawals`. After a withdrawal delay, a user can call `processWithdrawalQueue()`. This function does not allow a user to process the withdrawal based on an id but processes withdrawals in chronological order. When a user wants to withdraw funds, all other withdrawals that were initiated before need to be processed first. This can be problematic if one of the withdrawals fails for an unforeseen reason because then, the withdrawal queue is stuck, and no other withdrawals after the failing one can take place. Such a scenario could occur when a user with LP tokens gets blacklisted and initiates a withdrawal. USDC is planned to be the quote asset, and it has a blacklist function that has been used in the past for various reasons.

Vulnerability Detail

`_transferQuote()` is called when a withdrawal is processed as part of `processWithdrawalQueue()`. It attempts a token transfer and expects it to succeed in line 1060. It has no mechanism to handle a failing transfer and skip the queue.

Impact

The withdrawal queue could become permanently stuck, and users will not be able to withdraw their funds anymore from the `LiquidityPool` contract.

Code Snippet

```
_transferQuote(current.beneficiary, quoteAmount);
```

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L428

```
function _transferQuote(address to, uint amount) internal {
  amount = ConvertDecimals.convertFrom18(amount, quoteAsset.decimals());
  if (amount > 0) {
    if (!quoteAsset.transfer(to, amount)) {
      revert QuoteTransferFailed(address(this), address(this), to, amount);
    }
  }
}
```

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L1057-L1064

Tool used

Manual Review

Recommendation

The `transfer` call should be wrapped into a try/catch statement. If the `transfer` call in `processWithdrawalQueue()` fails, then the withdrawal should be skipped.

Discussion

hrishibhat

Sponsor comment:

> Valid, would even class this as high

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/6f0ee00187ea91eb92dcfa090fb6c398b8b51822

thec00n

`_tryTransferQuote()` catches potential exception from `transfer()` call and mints LP tokens again for the user, so reverting to the state before withdrawal started. (see tests https://github.com/lyra-finance/lyra-protocol/blob/997a65f8f7f24071882e71254a1863c19626e341/test/contracts/LiquidityPool/5_ProcessWithdrawal.ts#L392-L503)

thec00n

LGTM

Issue H-5: Multiple update position requests can be created with`updateCollateral()`

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/1

Found by

hyh, thec00n, clems4ever, ctf_sec

Summary

The `updateCollateral()` function adjusts positions on the GMX `PostionRouter` to match the target leverage of the liquidity pool. The function should only adjust positions when there is no pending order. Otherwise, the `pendingOrderKey` will be overwritten, making it impossible to cancel previous pending orders with `cancelPendingOrder()`. Also, it's possible to create multiple pending orders at the same time by calling `updateCollateral()` repeatedly.

Vulnerability Detail

When there are two positions open at the same time, then `updateCollateral()` can be called repeatedly to create update position requests. The check if position requests are allowed is only performed in `_getCurrentLeverage()` and checked on line 305.

Impact

A malicious user could call `updateCollateral()` repeatedly and create many update position requests in case there are two positions open at the same time. This could unbalance the overall hedging position to the point where it gets liquidated.

Code Snippet

```
function updateCollateral() external payable virtual override nonReentrant {
    CurrentPositions memory positions = _getPositions();
    emit HedgerPosition(positions);

    if (positions.amountOpen > 1) {
      int expectedHedge = _getCappedExpectedHedge();
      _closeSecondPosition(positions, expectedHedge);
      return;
    }
    (, bool needUpdate, int collateralDelta) = _getCurrentLeverage(positions);
    if (!needUpdate) return;
```

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L294-L305

Tool used

Manual Review

Recommendation

`updateCollateral()` should check if there are pending orders before creating any new requests on the GMX `PositionRouter` contract. So the pending order check should occur before `_closeSecondPosition()` is called.

Discussion

hrishibhat

Sponsor comment:

> Valid

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/6f0ee00187ea91eb92dcfa090fb6c398b8b51822

thec00n

Call `_hasPendingPositionRequest()` first in `updateCollateral()` and revert if there are pending position requests (see https://github.com/lyra-finance/lyra-protocol/blob/83de4d893a61e6728ed776763e66a1c1d99ce8a9/contracts/GMXFuturesPoolHedger.sol#L334-L338). LGTM.

Issue M-1: GMXFuturesPoolHedger uses payable.transfer calls with an arbitrary receiver

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/105

Found by

hyh, thec00n, ctf_sec, hansfriese

Summary

ETH transfers are done with `payable(msg.sender).transfer`, which can malfunction for smart contract receivers.

Vulnerability Detail

This is unsafe as `transfer` has hard coded gas budget and can fail when `msg.sender` is a smart contract. Such transactions will fail for smart contract users which don't fit to 2300 gas stipend `transfer` have.

The issues with `transfer()` are outlined here:

https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Impact

Funds will be unattainable for smart contract receivers.

Code Snippet

`payable(msg.sender).transfer` is used for an arbitrary `msg.sender`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L284-L287

```
    _hedgeDelta(expectedHedge);
    // return any excess eth
    payable(msg.sender).transfer(address(this).balance);
  }
```

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L329-L332

```
    emit CollateralOrderPosted(pendingOrderKey, positions.isLong, collateralDelta);
    // return any excess eth
```

```
  payable(msg.sender).transfer(address(this).balance);
}
```

Tool used

Manual Review

Recommendation

The recommendation is to use low-level `call.value(amount)` with the corresponding result check or employ OpenZeppelin's `Address.sendValue`:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol#L60

Discussion

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/6f0ee00187ea91eb92dcfa090fb6c398b8b51822

thec00n

All `transfer()` functions are replaced by `_returnAllEth()` which uses a low-level `call.value(amount)`. LGTM

GalloDaSballo

Escalate for 1 USDC

Disputing the findings severity because it doesn't apply to a risk that the in-scope codebase is taking, but rather a gotcha for integrators.

Specifically for three reasons:

1) The usage of .transfer will not break the majority of contracts (e.g. Gnosis Safe works fine)

2) The risk doesn't apply to the protocol, but to an undefined integrator, the security scope cannot include an imaginary contract that reverts

3) The suggestion is a best practice, the idea that this is a Medium Severity stems from the difference in how severities are classified in Contests (this one) vs other Audits, in which Medium Severity is used to flag potential risks rather than actual vulnerabilities.

To expand on this point here's the infamous Rari Capital Audit: https://consensys.net/diligence/audits/2021/01/fei-protocol/#unchecked-return-value-for-iwethtransfer-call

And the explanation for Medium Severity:
https://consensys.net/diligence/audits/2021/01/fei-protocol/#findings

Which I'll quote: `Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.`

This is in stark contrast to the severity classification used in this contest.

In conclusion I believe there's no objective way for this to be of Medium Severity, and recommend a Low / Informational Severity per the classification offered in the rules.

As a side note, the usage of `.call` in an unbounded way, opens up to additionally griefing attacks, which can also be used for DOSsing, see this document:
https://www.getsecureworld.com/blog/smart-contract-gas-griefing-attack-the-hidden-danger/

Meaning the mitigation will still need to cap the gas sent back to the caller

sherlock-admin

> Escalate for 1 USDC
>
> Disputing the findings severity because it doesn't apply to a risk that the in-scope codebase is taking, but rather a gotcha for integrators.
>
> Specifically for three reasons:
>
> 1) The usage of .transfer will not break the majority of contracts (e.g. Gnosis Safe works fine)
> 2) The risk doesn't apply to the protocol, but to an undefined integrator, the security scope cannot include an imaginary contract that reverts
> 3) The suggestion is a best practice, the idea that this is a Medium Severity stems from the difference in how severities are classified in Contests (this one) vs other Audits, in which Medium Severity is used to flag potential risks rather than actual vulnerabilities.
>
> To expand on this point here's the infamous Rari Capital Audit:
> https://consensys.net/diligence/audits/2021/01/fei-protocol/#unchecked-return-value-for-iwethtransfer-call
>
> And the explanation for Medium Severity:
> https://consensys.net/diligence/audits/2021/01/fei-protocol/#findings
>
> Which I'll quote: `Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.`
>
> This is in stark contrast to the severity classification used in this contest.

> In conclusion I believe there's no objective way for this to be of Medium Severity, and recommend a Low / Informational Severity per the classification offered in the rules.
>
> As a side note, the usage of `.call` in an unbounded way, opens up to additionally griefing attacks, which can also be used for DOSsing, see this document: https://www.getsecureworld.com/blog/smart-contract-gas-griefing-attack-the-hidden-danger/
>
> Meaning the mitigation will still need to cap the gas sent back to the caller

You've created a valid escalation for 1 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

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

Issue M-2: longScaleFactor for an expired board can be manipulated by back running big initiateDeposit

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/103

Found by

hyh

Summary

NAV can be increased as `totalQueuedDeposits` can be dropped down via processDepositQueue() right before settleExpiredBoard() call.

Bob the board payoff beneficiary can atomically run processDepositQueue(), then settleExpiredBoard() (both are public) right after big new deposit request to enhance the resulting NAV that will be used for `longScaleFactor` calculation of his board.

Vulnerability Detail

As a result of processDepositQueue() NAV will be increased by the full processed deposit value, `processedDeposits`, but will be decreased only by `DecimalMath.UNIT - lpParams.adjustmentNetScalingFactor` part of it.

Bob can back-run large initiateDeposit() calls with such a settlement. Bob's board will take advantage from local increase of liquidity conditions as the expense of other board settlements who do not time their settleExpiredBoard() calls.

Impact

Suppose deposits and withdrawals have some distribution reasonably close to random. By picking the right moment Bob's board will gain artificially bigger `longScaleFactor` and payouts that are linked to it, at the expense of other boards. I.e. say there is one deposit and one withdrawal, Bob back-runs the deposit and gained the corresponding payout boost. Other board will be settled in a less favourable time, say around withdrawal, and will obtain lesser payout just by the sake of this timing.

For the beneficiaries of the settlements of distinct boards it's a kind of zero sum game and the winning scenario can be forced this way at the expense of all other parties.

The impact is other boards lose money as Bob has indirectly stolen from them. In absence of any significant prerequisites setting the severity to be high.

Code Snippet

processDepositQueue() increases NAV by `processedDeposits` and decreases it by `(DecimalMath.UNIT - lpParams.adjustmentNetScalingFactor) * processedDeposits` by adding it to `protectedQuote`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L392-L399

```
// only update if deposit processed to avoid changes when CB's are firing
if (processedDeposits != 0) {
  totalQueuedDeposits -= processedDeposits;

  protectedQuote = (liquidity.NAV + processedDeposits).multiplyDecimal(
    DecimalMath.UNIT - lpParams.adjustmentNetScalingFactor
  );
}
```

NAV is determined off live LiquidityPool and GMX (`usedDeltaLiquidity`) NPV of the assets less `protectedQuote`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L898-L921

```
function _getTotalPoolValueQuote(
  uint basePrice,
  uint usedDeltaLiquidity,
  int optionValueDebt
) internal view returns (uint, uint) {
  int totalAssetValue = SafeCast.toInt256(
    ConvertDecimals.convertTo18(quoteAsset.balanceOf(address(this)),
↪   quoteAsset.decimals()) +
      ConvertDecimals.convertTo18(baseAsset.balanceOf(address(this)),
↪   baseAsset.decimals()).multiplyDecimal(basePrice)
  ) +
    SafeCast.toInt256(usedDeltaLiquidity) -
    SafeCast.toInt256(totalOutstandingSettlements + totalQueuedDeposits);

  if (totalAssetValue < 0) {
    revert NegativeTotalAssetValue(address(this), totalAssetValue);
  }

  // If debt is negative we can simply return TAV - (-debt)
  // availableAssetValue here is +'ve and optionValueDebt is -'ve so we can
↪   safely return uint
  if (optionValueDebt < 0) {
    return (SafeCast.toUint256(totalAssetValue - optionValueDebt),
↪   DecimalMath.UNIT);
```

```
  }

  // ensure a percentage of the pool's NAV is always protected from AMM's
↪  insolvency
  int availableAssetValue = totalAssetValue - int(protectedQuote);
```

`longScaleFactor` is determined via liquidity as of closing time:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L670-L699

```
function boardSettlement(
  uint insolventSettlements,
  uint amountQuoteFreed,
  uint amountQuoteReserved,
  uint amountBaseFreed
) external onlyOptionMarket returns (uint) {
  // Update circuit breaker whenever a board is settled, to pause
↪  deposits/withdrawals
  // This allows keepers some time to settle insolvent positions
  if (block.timestamp + cbParams.boardSettlementCBTimeout > CBTimestamp) {
    CBTimestamp = block.timestamp + cbParams.boardSettlementCBTimeout;
    emit BoardSettlementCircuitBreakerUpdated(CBTimestamp);
  }

  insolventSettlementAmount += insolventSettlements;

  _freePutCollateral(amountQuoteFreed);
  _freeCallCollateral(amountBaseFreed);

  // If amountQuoteReserved > available liquidity, amountQuoteReserved is scaled
↪  down to an available amount
  Liquidity memory liquidity = getLiquidity(); // calculates total pool value
↪  and potential scaling

  totalOutstandingSettlements +=
↪  amountQuoteReserved.multiplyDecimal(liquidity.longScaleFactor);

  emit BoardSettlement(insolventSettlementAmount, amountQuoteReserved,
↪  totalOutstandingSettlements);

  if (address(poolHedger) != address(0)) {
    poolHedger.resetInteractionDelay();
  }
  return liquidity.longScaleFactor;
}
```

It is then recorded to `scaledLongsForBoard` in _settleExpiredBoard():

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/OptionMarket.sol#L1051-L1104

```
function _settleExpiredBoard(OptionBoard memory board) internal {
  uint spotPrice = exchangeAdapter.getSettlementPriceForMarket(address(this),
↪   board.expiry);
  ...
  (uint lpBaseInsolvency, uint lpQuoteInsolvency) =
↪   shortCollateral.boardSettlement(
    totalAMMShortCallProfitBase,
    totalAMMShortPutProfitQuote + totalAMMShortCallProfitQuote
  );

  // This will batch all base we want to convert to quote and sell it in one
↪   transaction
  uint longScaleFactor = liquidityPool.boardSettlement(
    lpQuoteInsolvency + lpBaseInsolvency.multiplyDecimal(spotPrice),
    totalBoardLongPutCollateral,
    totalUserLongProfitQuote,
    totalBoardLongCallCollateral
  );
  scaledLongsForBoard[board.id] = longScaleFactor;
```

Then used in settlements:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/ShortCollateral.sol#L179-L214

```
function settleOptions(uint[] memory positionIds) external nonReentrant
↪   notGlobalPaused {
  // This is how much is missing from the ShortCollateral contract that was
↪   claimed by LPs at board expiry
  // We want to take it back when we know how much was missing.
  uint baseInsolventAmount = 0;
  uint quoteInsolventAmount = 0;

  OptionToken.PositionWithOwner[] memory optionPositions =
↪   optionToken.getPositionsWithOwner(positionIds);
  optionToken.settlePositions(positionIds);

  uint positionsLength = optionPositions.length;
  for (uint i = 0; i < positionsLength; ++i) {
    OptionToken.PositionWithOwner memory position = optionPositions[i];
    uint settlementAmount = 0;
    uint insolventAmount = 0;
    (uint strikePrice, uint priceAtExpiry, uint ammShortCallBaseProfitRatio,
↪   uint longScaleFactor) = optionMarket
      .getSettlementParameters(position.strikeId);

    if (priceAtExpiry == 0) {
      revert BoardMustBeSettled(address(this), position);
    }

    if (position.optionType == OptionMarket.OptionType.LONG_CALL) {
      settlementAmount = _sendLongCallProceeds(
        position.owner,
        position.amount.multiplyDecimal(longScaleFactor),
        strikePrice,
        priceAtExpiry
      );
    } else if (position.optionType == OptionMarket.OptionType.LONG_PUT) {
      settlementAmount = _sendLongPutProceeds(
        position.owner,
        position.amount.multiplyDecimal(longScaleFactor),
        strikePrice,
        priceAtExpiry
      );
    }
```

This way Bob have direct payoff boost from the `longScaleFactor` increase.

Tool used

Manual Review

Recommendation

Notice that withdrawal processing have similar, but opposite effect (although a bit more complicated due to token price also depending on NAV): some amount is removed from the balance, but only part of it is cleared from `protectedQuote`, so NAV decreases. For example, it can be used for griefing in a similar setting.

One of the possible approaches might be restricting both processDepositQueue() and processWithdrawalQueue() to be run by a protocol controlled keeper script only. There a condition for it to be run might be both time and deposit-withdrawal balance dependent, i.e., as an example, it can run not less frequently that once per 24h, but tend to run in a situations when total amounts of deposits and withdrawals are reasonably close to minimize the NAV impact.

Discussion

hrishibhat

Sponsor comment:

> Invalid - deposits and withdrawals are blocked (besides guardian) whenever longScaleFactor is not 1 via the circuit breaker. Thus the value can't really be manipulated by large deposits.

dmitriia

Escalate for 80 USDC

The attack vector looks to be possible despite CB when `liquidity.longScaleFactor == DecimalMath.UNIT` via withdrawal processing as mentioned in the Recommendation section.

I.e. an attacker will call processWithdrawalQueue() (the corresponding withdrawals will not be processed right away when `optionMarket.getNumLiveBoards() > 0`) before board settlement and have `NAV` reduced. Net impact will be griefing as `longScaleFactor` as a result can drop below 1 with the corresponding effect on board payouts and loss for the beneficiaries.

The issue has similar nature to #19, i.e. different components of NAV can be manipulated by calling public protocol management functions. There it is updateCollateral(), here it is processWithdrawalQueue().

The withdrawals are put to the queue when `optionMarket.getNumLiveBoards() > 0`:

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/LiquidityPool.sol#L308-L340

```
function initiateWithdraw(address beneficiary, uint amountLiquidityToken)
↪  external nonReentrant {
  ...

  if (optionMarket.getNumLiveBoards() == 0 && liquidity.longScaleFactor ==
↪  DecimalMath.UNIT) {
    _transferQuote(beneficiary, withdrawalValue);

    protectedQuote = (liquidity.NAV - withdrawalValue).multiplyDecimal(
      DecimalMath.UNIT - lpParams.adjustmentNetScalingFactor
    );

    // quoteReceived in the event is in 18dp
    emit WithdrawProcessed(
      msg.sender,
      beneficiary,
      0,
      amountLiquidityToken,
      tokenPrice,
      withdrawalValue,
      totalQueuedWithdrawals,
      block.timestamp
    );
  } else {
    QueuedWithdrawal storage newWithdrawal =
↪   queuedWithdrawals[nextQueuedWithdrawalId];
```

Processing the withdrawal decreases the NAV as the whole amount withdrawn is sent out from the balance, while only part of it is being removed from `protectedQuote`:

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/LiquidityPool.sol#L402-L462

```
/// @param limit number of withdrawal tickets to process in a single transaction
↪   to avoid gas limit soft-locks
function processWithdrawalQueue(uint limit) external nonReentrant {
  ...
    _transferQuote(current.beneficiary, quoteAmount);
  ...
  // only update if withdrawal processed to avoid changes when CB's are firing
  // getLiquidity() called again to account for withdrawal fee
  if (oldQueuedWithdrawals > totalQueuedWithdrawals) {
    Liquidity memory liquidity = getLiquidity();
    protectedQuote = liquidity.NAV.multiplyDecimal(DecimalMath.UNIT -
↪   lpParams.adjustmentNetScalingFactor);
  }
}
```

This way an attacker can monitor the situation and catch the moment when there is a just expired board and liquidity situation barely made it to the full repayment and there is a big enough withdrawal queued. By pushing the processing forward and immediately settling the board, the attacker will force the long option holders into losses.

sherlock-admin

> Escalate for 80 USDC
>
> The attack vector looks to be possible despite CB when `liquidity.longScaleFactor == DecimalMath.UNIT` via withdrawal processing as mentioned in the Recommendation section.
>
> I.e. an attacker will call processWithdrawalQueue() (the corresponding withdrawals will not be processed right away when `optionMarket.getNumLiveBoards() > 0`) before board settlement and have `NAV` reduced. Net impact will be griefing as `longScaleFactor` as a result can drop below 1 with the corresponding effect on board payouts and loss for the beneficiaries.

The issue has similar nature to #19, i.e. different components of NAV can be manipulated by calling public protocol management functions. There it is updateCollateral(), here it is processWithdrawalQueue().

The withdrawals are put to the queue when `optionMarket.getNumLiveBoards() > 0`:

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/LiquidityPool.sol#L308-L340

```
function initiateWithdraw(address beneficiary, uint amountLiquidityToken)
↪  external nonReentrant {
  ...

  if (optionMarket.getNumLiveBoards() == 0 && liquidity.longScaleFactor ==
↪  DecimalMath.UNIT) {
    _transferQuote(beneficiary, withdrawalValue);

    protectedQuote = (liquidity.NAV - withdrawalValue).multiplyDecimal(
      DecimalMath.UNIT - lpParams.adjustmentNetScalingFactor
    );

    // quoteReceived in the event is in 18dp
    emit WithdrawProcessed(
      msg.sender,
      beneficiary,
      0,
      amountLiquidityToken,
      tokenPrice,
      withdrawalValue,
      totalQueuedWithdrawals,
      block.timestamp
    );
  } else {
    QueuedWithdrawal storage newWithdrawal =
↪  queuedWithdrawals[nextQueuedWithdrawalId];
```

Processing the withdrawal decreases the NAV as the whole amount withdrawn is sent out from the balance, while only part of it is being removed from `protectedQuote`:

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/LiquidityPool.sol#L402-L462

```
/// @param limit number of withdrawal tickets to process in a single
↪  transaction to avoid gas limit soft-locks
```

```
function processWithdrawalQueue(uint limit) external nonReentrant {
  ...
    _transferQuote(current.beneficiary, quoteAmount);
  ...
  // only update if withdrawal processed to avoid changes when CB's are
↪  firing
  // getLiquidity() called again to account for withdrawal fee
  if (oldQueuedWithdrawals > totalQueuedWithdrawals) {
    Liquidity memory liquidity = getLiquidity();
    protectedQuote = liquidity.NAV.multiplyDecimal(DecimalMath.UNIT -
↪  lpParams.adjustmentNetScalingFactor);
  }
}
```

This way an attacker can monitor the situation and catch the moment when there is a just expired board and liquidity situation barely made it to the full repayment and there is a big enough withdrawal queued. By pushing the processing forward and immediately settling the board, the attacker will force the long option holders into losses.

You've created a valid escalation for 80 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

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

0xdomrom

I'm genuinely not understanding the scenario... can a test be written up to mimic the scenario? That may help.

One comment to the above: circuit breakers apply to both deposits and withdrawals - and there are circuit breakers when boards are settled, precisely to make sure insolvent positions can be settled in time before desposits and withdrawals are re-enabled.

dmitriia

This actually is pretty straightforward: as processing a withdrawal decreases NAV by removing quote funds from the balance, then in a borderline situations, i.e. when available liquidity is barely covering `optionValueDebt`, but `longScaleFactor` is still equal to 1, processing a withdrawal just before board settlement is a griefing for its beneficiaries as it will put `longScaleFactor` under 1 and truncate the payouts.

POC:

0. Let's say that `liquidity.longScaleFactor == DecimalMath.UNIT`,

`optionMarket.getNumLiveBoards() > 0.`

1. Alice is a big LP and just requested a withdrawal. It will be put to a queue. Let's say there is nothing else in deposit and withdrawal queues.
2. A board '10' is just expired and OptionMarket's `settleExpiredBoard(10)` -> LiquidityPool's boardSettlement() can be run for it.
3. Bob the attacker can call LiquidityPool's `processWithdrawalQueue(1)` and then immediately `settleExpiredBoard(10)`.
4. If Bob were not present, say a keeper or Mike the beneficiary will just call `settleExpiredBoard(10)` while Alice's withdrawal request is still in the queue, and have receive full payouts with `longScaleFactor` of 1.
5. As Bob's `processWithdrawalQueue(1)` reduced the NAV by sending quote funds to Alice (`transferQuote(current.beneficiary, quoteAmount)` in processWithdrawalQueue()), while in _getTotalPoolValueQuote() `availableAssetValue` are reduced by `lpParams.adjustmentNetScalingFactor` of these funds (say it's `0.9` as https://leaps.lyra.finance/leaps/leap-36/ goes), so, as before that `availableAssetValue` just barely covered the `optionValueDebt`, now it's a deficit and `longScaleFactor` of the `10` board will be printed below 1.

LiquidityPool's NAV calculation:

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/LiquidityPool.sol#L898-L940

```
function _getTotalPoolValueQuote(
  uint basePrice,
  uint usedDeltaLiquidity,
  int optionValueDebt
) internal view returns (uint, uint) {
  int totalAssetValue = SafeCast.toInt256(
    ConvertDecimals.convertTo18(quoteAsset.balanceOf(address(this)),
↪   quoteAsset.decimals()) +
      ConvertDecimals.convertTo18(baseAsset.balanceOf(address(this)),
↪   baseAsset.decimals()).multiplyDecimal(basePrice)
  ) +
    SafeCast.toInt256(usedDeltaLiquidity) -
    SafeCast.toInt256(totalOutstandingSettlements + totalQueuedDeposits);

  if (totalAssetValue < 0) {
    revert NegativeTotalAssetValue(address(this), totalAssetValue);
  }

  // If debt is negative we can simply return TAV - (-debt)
  // availableAssetValue here is +'ve and optionValueDebt is -'ve so we can
↪ safely return uint
  if (optionValueDebt < 0) {
    return (SafeCast.toUint256(totalAssetValue - optionValueDebt),
↪   DecimalMath.UNIT);
  }

  // ensure a percentage of the pool's NAV is always protected from AMM's
↪   insolvency
  int availableAssetValue = totalAssetValue - int(protectedQuote);
  uint longScaleFactor = DecimalMath.UNIT;

  // in extreme situations, if the TAV < reserved cash, set long options to
↪   worthless
  if (availableAssetValue < 0) {
    return (SafeCast.toUint256(totalAssetValue), 0);
  }

  // NOTE: the longScaleFactor is calculated using the total option debt however
↪   only the long debts are scaled down
  // when paid out. Therefore the asset value affected is less than the real
↪   amount.
  if (availableAssetValue < optionValueDebt) {
    // both guaranteed to be positive
    longScaleFactor = SafeCast.toUint256(availableAssetValue).divideDecimal(Safe ⌋
↪   Cast.toUint256(optionValueDebt));
  }

  return (
    SafeCast.toUint256(totalAssetValue) -
↪   SafeCast.toUint256(optionValueDebt).multiplyDecimal(longScaleFactor),
    longScaleFactor
  );
}
```

processWithdrawalQueue() will reduce _getTotalPoolValueQuote()'s `totalAssetValue` by `quoteAmount` with `_transferQuote(current.beneficiary, quoteAmount)`, while reduce `availableAssetValue` by `lpParams.adjustmentNetScalingFactor` (90%) of it:

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/LiquidityPool.sol#L402-L462

```
/// @param limit number of withdrawal tickets to process in a single transaction
↪   to avoid gas limit soft-locks
function processWithdrawalQueue(uint limit) external nonReentrant {
  ...

    uint quoteAmount = burnAmount.multiplyDecimal(tokenPriceWithFee);
    current.quoteSent += quoteAmount;
    _transferQuote(current.beneficiary, quoteAmount);
    ...

  // only update if withdrawal processed to avoid changes when CB's are firing
  // getLiquidity() called again to account for withdrawal fee
  if (oldQueuedWithdrawals > totalQueuedWithdrawals) {
    Liquidity memory liquidity = getLiquidity();
    protectedQuote = liquidity.NAV.multiplyDecimal(DecimalMath.UNIT -
↪   lpParams.adjustmentNetScalingFactor);
  }
}
```

Board's `longScaleFactor` is calculated on settlement and cannot be changed thereafter:

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/OptionMarket.sol#L1018-L1104

```
function settleExpiredBoard(uint boardId) external nonReentrant {
  OptionBoard memory board = optionBoards[boardId];
  if (board.id != boardId || board.id == 0) {
    revert InvalidBoardId(address(this), boardId);
  }
  if (block.timestamp < board.expiry) {
    revert BoardNotExpired(address(this), boardId);
  }
  _clearAndSettleBoard(board);
}

function _clearAndSettleBoard(OptionBoard memory board) internal {
  ...

  _settleExpiredBoard(board);
  greekCache.removeBoard(board.id);
}

function _settleExpiredBoard(OptionBoard memory board) internal {
  ...

  // This will batch all base we want to convert to quote and sell it in one
↪   transaction
  uint longScaleFactor = liquidityPool.boardSettlement(
    lpQuoteInsolvency + lpBaseInsolvency.multiplyDecimal(spotPrice),
    totalBoardLongPutCollateral,
    totalUserLongProfitQuote,
    totalBoardLongCallCollateral
  );
  scaledLongsForBoard[board.id] = longScaleFactor;
```

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/LiquidityPool.sol#L670-L699

```
function boardSettlement(
  uint insolventSettlements,
  uint amountQuoteFreed,
  uint amountQuoteReserved,
  uint amountBaseFreed
) external onlyOptionMarket returns (uint) {
  // Update circuit breaker whenever a board is settled, to pause
↪   deposits/withdrawals
  // This allows keepers some time to settle insolvent positions
  if (block.timestamp + cbParams.boardSettlementCBTimeout > CBTimestamp) {
    CBTimestamp = block.timestamp + cbParams.boardSettlementCBTimeout;
    emit BoardSettlementCircuitBreakerUpdated(CBTimestamp);
  }

  insolventSettlementAmount += insolventSettlements;

  _freePutCollateral(amountQuoteFreed);
  _freeCallCollateral(amountBaseFreed);

  // If amountQuoteReserved > available liquidity, amountQuoteReserved is scaled
↪   down to an available amount
  Liquidity memory liquidity = getLiquidity(); // calculates total pool value
↪   and potential scaling

  totalOutstandingSettlements +=
↪   amountQuoteReserved.multiplyDecimal(liquidity.longScaleFactor);

  emit BoardSettlement(insolventSettlementAmount, amountQuoteReserved,
↪   totalOutstandingSettlements);

  if (address(poolHedger) != address(0)) {
    poolHedger.resetInteractionDelay();
  }
  return liquidity.longScaleFactor;
}
```

It is LP vs long options holders payouts, the issue is that LP can manipulate in their favor by front running the settlement with a withdrawal in such borderline cases. Say Bob can do that on behalf of Alice.

Can't see the nicer solution here, but scheduling the deposit/withdrawal processing along with adding access controls to it can remove the possibility.

0xdomrom

Appreciate the extra effort here, I'm pretty sure I understand it now

but the 0.9 value you mention means 10% of the value post withdrawal is the "reserved amount", and the reserved amount is shrinking. I'm really not seeing this as a likely (or even realistically possible) scenario. Again, writing a short test to recreate this scenario would be super helpful

I've tried here but was unable to recreate the exact example you are trying to explain. I would appreciate it if you could edit this to make it work.

```
import {deployFixture} from '../../utils/fixture';
import {expect, hre} from "../../utils/testSetup";
import {HOUR_SEC, MAX_UINT, MONTH_SEC, OptionType, toBN} from
↪  "../../../scripts/util/web3utils";
import {createDefaultBoardWithOverrides, mockPrice} from
↪  "../../utils/seedTestSystem";
import {openPosition} from "../../utils/contractHelpers";
import {DEFAULT_CB_PARAMS, DEFAULT_LIQUIDITY_POOL_PARAMS} from
↪  "../../utils/defaultParams";
import {fastForward, restoreSnapshot, takeSnapshot} from "../../utils/evm";

// Do full integration tests here (e.g. open trades/make deposits/hedge delta)
describe('Liquidity Accounting', async () => {
  beforeEach(deployFixture);

  it.only('tests edge case', async () => {
    await mockPrice(hre.f.c, toBN('1000'), 'sETH');
    await hre.f.c.liquidityPool.setLiquidityPoolParameters({
      ...DEFAULT_LIQUIDITY_POOL_PARAMS,
      // NOTE: setting this to reserve 90% of the liquidity instead of just 10%
↪   to make the edge case more easily hittable
      adjustmentNetScalingFactor: toBN('0.1'),
      callCollatScalingFactor: toBN('0.7'),
      // NOTE: also withdrawal fee set to 0 for this example, which also
↪   mitigates the edge case being tested
      withdrawalFee: 0
    })
    const bob = hre.f.signers[2];
    await hre.f.c.snx.quoteAsset.mint(hre.f.deployer.address, toBN('1000000'));
    await hre.f.c.snx.quoteAsset.mint(hre.f.alice.address, toBN('1000'));
    await hre.f.c.snx.quoteAsset.mint(bob.address, toBN('100000'));

    await hre.f.c.snx.quoteAsset.connect(hre.f.deployer).approve(hre.f.c.liquidi⌋
↪   tyPool.address, MAX_UINT);
    await hre.f.c.snx.quoteAsset.connect(hre.f.alice).approve(hre.f.c.liquidityP⌋
↪   ool.address, MAX_UINT);
```

```
    await
↪   hre.f.c.snx.quoteAsset.connect(bob).approve(hre.f.c.optionMarket.address,
↪   MAX_UINT);

    await hre.f.c.liquidityPool.initiateDeposit(hre.f.deployer.address,
↪   toBN('1000000'));
    await hre.f.c.liquidityPool.connect(hre.f.alice).initiateDeposit(hre.f.alice ⌋
↪   .address, toBN('1000'));

    expect((await
↪   hre.f.c.liquidityPool.getLiquidity()).freeLiquidity).eq(toBN('1001000'));

    const board1 = await createDefaultBoardWithOverrides(hre.f.c, {expiresIn:
↪   MONTH_SEC, strikePrices: ['1000'], skews: ['1']});
    await createDefaultBoardWithOverrides(hre.f.c, {expiresIn: MONTH_SEC * 2,
↪   strikePrices: ['1000'], skews: ['1']});
    const boardStrikes = await
↪   hre.f.c.optionMarketViewer.getBoard(hre.f.c.optionMarket.address, board1);

    await openPosition({
      amount: toBN('0.85'),
      optionType: OptionType.LONG_CALL,
      strikeId: boardStrikes.strikes[0].strikeId
    }, bob, hre.f.c);

    await hre.f.c.liquidityPool.connect(hre.f.deployer).initiateWithdraw(hre.f.d ⌋
↪   eployer.address, toBN('1000000'));

    console.log(await hre.f.c.liquidityPool.getLiquidity());

    // With default parameters:
    // As free liquidity is < 1% of total NAV, circuit breaker is fired and
↪   withdrawal can't go through
    {
      const snapshot = await takeSnapshot();
      await hre.f.c.liquidityPool.updateCBs();
      expect(await hre.f.c.liquidityPool.CBTimestamp()).gt(0);
      await restoreSnapshot(snapshot);
    }

    // So let's ignore that protection for now, to test the edge case
    await hre.f.c.liquidityPool.setCircuitBreakerParameters({
      ...DEFAULT_CB_PARAMS,
      liquidityCBThreshold: 0,
    })
    await hre.f.c.liquidityPool.updateCBs();
    expect(await hre.f.c.liquidityPool.CBTimestamp()).eq(0);
    await fastForward(MONTH_SEC - HOUR_SEC);

    await hre.f.c.keeperHelper.updateAllBoardCachedGreeks();

    await mockPrice(hre.f.c, toBN('1500'), 'sETH');
    {
      const snapshot = await takeSnapshot();

      // CASE 1, cache is updated with price
      await hre.f.c.keeperHelper.updateAllBoardCachedGreeks();
      const tx = await hre.f.c.liquidityPool.processWithdrawalQueue(1);
      console.log((await tx.wait()).events)

      await fastForward(HOUR_SEC);
      await hre.f.c.optionMarket.settleExpiredBoard(board1);

      // withdrawal head wasn't fully withdrawn (only partial)
      console.log(await hre.f.c.liquidityPool.queuedWithdrawals(await
  ↪ hre.f.c.liquidityPool.queuedWithdrawalHead()));

      await restoreSnapshot(snapshot);
    }

    {
      const snapshot = await takeSnapshot();

      // CASE 2, cache isn't updated first
      await hre.f.c.liquidityPool.processWithdrawalQueue(1);

      await hre.f.c.keeperHelper.updateAllBoardCachedGreeks();
      await fastForward(HOUR_SEC);
      await hre.f.c.optionMarket.settleExpiredBoard(board1);

      // withdrawal head was fully withdrawn
      console.log(await hre.f.c.liquidityPool.queuedWithdrawals(await
  ↪ hre.f.c.liquidityPool.queuedWithdrawalHead()));
      console.log(await hre.f.c.optionMarket.getSettlementParameters(boardStrike
  ↪ s.strikes[0].strikeId));

      await restoreSnapshot(snapshot);
    }
  });
});
```

dmitriia

Updated the test case to match the example:

```
import {deployFixture} from '../../utils/fixture';
import {expect, hre} from "../../utils/testSetup";
import {HOUR_SEC, MAX_UINT, MONTH_SEC, OptionType, toBN} from
↪  "../../../scripts/util/web3utils";
import {createDefaultBoardWithOverrides, mockPrice} from
↪  "../../utils/seedTestSystem";
import {openPosition} from "../../utils/contractHelpers";
import {DEFAULT_CB_PARAMS, DEFAULT_LIQUIDITY_POOL_PARAMS} from
↪  "../../utils/defaultParams";
import {fastForward, restoreSnapshot, takeSnapshot} from "../../utils/evm";

// Do full integration tests here (e.g. open trades/make deposits/hedge delta)
describe('Liquidity Accounting', async () => {
  beforeEach(deployFixture);

  it.only('tests edge case', async () => {
    await mockPrice(hre.f.c, toBN('1000'), 'sETH');
    await hre.f.c.liquidityPool.setLiquidityPoolParameters({
      ...DEFAULT_LIQUIDITY_POOL_PARAMS,
      // NOTE: setting this to reserve 90% of the liquidity instead of just 10%
↪   to make the edge case more easily hittable
      adjustmentNetScalingFactor: toBN('0.1'),
      callCollatScalingFactor: toBN('0.7'),
      // NOTE: also withdrawal fee set to 0 for this example, which also
↪   mitigates the edge case being tested
      withdrawalFee: 0
    })
    const bob = hre.f.signers[2];
    await hre.f.c.snx.quoteAsset.mint(hre.f.deployer.address, toBN('1000000'));
    await hre.f.c.snx.quoteAsset.mint(hre.f.alice.address, toBN('1000'));
    await hre.f.c.snx.quoteAsset.mint(bob.address, toBN('100000'));

    await hre.f.c.snx.quoteAsset.connect(hre.f.deployer).approve(hre.f.c.liquidi ⌋
↪   tyPool.address, MAX_UINT);
    await hre.f.c.snx.quoteAsset.connect(hre.f.alice).approve(hre.f.c.liquidityP ⌋
↪   ool.address, MAX_UINT);
    await
↪   hre.f.c.snx.quoteAsset.connect(bob).approve(hre.f.c.optionMarket.address,
↪   MAX_UINT);

    await hre.f.c.liquidityPool.initiateDeposit(hre.f.deployer.address,
↪   toBN('1000000'));
    await hre.f.c.liquidityPool.connect(hre.f.alice).initiateDeposit(hre.f.alice ⌋
↪   .address, toBN('1000'));

    expect((await
↪   hre.f.c.liquidityPool.getLiquidity()).freeLiquidity).eq(toBN('1001000'));
```

```
  const board1 = await createDefaultBoardWithOverrides(hre.f.c, {expiresIn:
↪ MONTH_SEC, strikePrices: ['1000'], skews: ['1']});
  await createDefaultBoardWithOverrides(hre.f.c, {expiresIn: MONTH_SEC * 2,
↪ strikePrices: ['1000'], skews: ['1']});
  const boardStrikes = await
↪ hre.f.c.optionMarketViewer.getBoard(hre.f.c.optionMarket.address, board1);

  await openPosition({
    amount: toBN('0.85'),
    optionType: OptionType.LONG_CALL,
    strikeId: boardStrikes.strikes[0].strikeId
  }, bob, hre.f.c);

  await hre.f.c.liquidityPool.connect(hre.f.deployer).initiateWithdraw(hre.f.d
↪ eployer.address, toBN('1000000'));

  console.log(await hre.f.c.liquidityPool.getLiquidity());

  // With default parameters:
  // As free liquidity is < 1% of total NAV, circuit breaker is fired and
↪ withdrawal can't go through
  {
    const snapshot = await takeSnapshot();
    await hre.f.c.liquidityPool.updateCBs();
    expect(await hre.f.c.liquidityPool.CBTimestamp()).gt(0);
    await restoreSnapshot(snapshot);
  }

  // So let's ignore that protection for now, to test the edge case
  await hre.f.c.liquidityPool.setCircuitBreakerParameters({
    ...DEFAULT_CB_PARAMS,
    liquidityCBThreshold: 0,
  })
  await hre.f.c.liquidityPool.updateCBs();
  expect(await hre.f.c.liquidityPool.CBTimestamp()).eq(0);

  await fastForward(MONTH_SEC - HOUR_SEC);

  await hre.f.c.keeperHelper.updateAllBoardCachedGreeks();

  await mockPrice(hre.f.c, toBN('1500'), 'sETH');

  await hre.f.c.keeperHelper.updateAllBoardCachedGreeks();
  {
    const snapshot = await takeSnapshot();

    // CASE 1, withdrawal processing before board settlement
```

```
      //
      console.log("CASE 1: first withdrawal, second board settlement");
      await fastForward(HOUR_SEC);
      const tx = await hre.f.c.liquidityPool.processWithdrawalQueue(1);
      console.log((await tx.wait()).events)
      await hre.f.c.optionMarket.settleExpiredBoard(board1);

      // withdrawal head wasn't fully withdrawn (only partial)
      console.log(await hre.f.c.liquidityPool.queuedWithdrawals(await
↪    hre.f.c.liquidityPool.queuedWithdrawalHead()));
      console.log(await hre.f.c.optionMarket.getSettlementParameters(boardStrike
↪    s.strikes[0].strikeId));

      await restoreSnapshot(snapshot);
    }

    {
      const snapshot = await takeSnapshot();

      // CASE 2, board settlement, withdrawal isn't processed first
      console.log("CASE 2: no withdrawal, only board settlement");
      await fastForward(HOUR_SEC);
      await hre.f.c.optionMarket.settleExpiredBoard(board1);

      // withdrawal head was fully withdrawn
      console.log(await hre.f.c.liquidityPool.queuedWithdrawals(await
↪    hre.f.c.liquidityPool.queuedWithdrawalHead()));
      console.log(await hre.f.c.optionMarket.getSettlementParameters(boardStrike
↪    s.strikes[0].strikeId));

      await restoreSnapshot(snapshot);
    }
  });
});
```

Output (withdrawal was processed, `longScaleFactor` became lower than 1 due to that):

```
CASE 1: first withdrawal, second board settlement
[
  BigNumber { value: "1" },
  '0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266',
  BigNumber { value: "7430540047103050854448" },
  BigNumber { value: "998936572922427183032635" },
  BigNumber { value: "1673967652" },
  id: BigNumber { value: "1" },
  beneficiary: '0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266',
  amountTokens: BigNumber { value: "743054004710305085448" },
  quoteSent: BigNumber { value: "998936572922427183032635" },
  withdrawInitiatedTime: BigNumber { value: "1673967652" }
]
[
  BigNumber { value: "1000000000000000000000" },
  BigNumber { value: "1500000000000000000000" },
  BigNumber { value: "335852225020990763" },
  BigNumber { value: "509994194336798955" },
  strikePrice: BigNumber { value: "1000000000000000000000" },
  priceAtExpiry: BigNumber { value: "1500000000000000000000" },
  strikeToBaseReturned: BigNumber { value: "335852225020990763" },
  longScaleFactor: BigNumber { value: "509994194336798955" }
]
CASE 2: no withdrawal, only board settlement
[
  BigNumber { value: "1" },
  '0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266',
  BigNumber { value: "1000000000000000000000000" },
  BigNumber { value: "0" },
  BigNumber { value: "1673967652" },
  id: BigNumber { value: "1" },
  beneficiary: '0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266',
  amountTokens: BigNumber { value: "1000000000000000000000000" },
  quoteSent: BigNumber { value: "0" },
  withdrawInitiatedTime: BigNumber { value: "1673967652" }
]
[
  BigNumber { value: "1000000000000000000000" },
  BigNumber { value: "1500000000000000000000" },
  BigNumber { value: "335852225020990763" },
  BigNumber { value: "1000000000000000000" },
  strikePrice: BigNumber { value: "1000000000000000000000" },
  priceAtExpiry: BigNumber { value: "1500000000000000000000" },
  strikeToBaseReturned: BigNumber { value: "335852225020990763" },
  longScaleFactor: BigNumber { value: "1000000000000000000" }
]
```

0xdomrom

Yep, thanks! Will include this in the test suite. As the test shows, there are several protections in place to prevent this causing an issue - and even with them removed, it takes a really specific scenario for this to actually have an effect. As such, I'd class this as low

dmitriia

Due to the specific focus on fund loss in Sherlock methodology Medium is exactly low/medium probability scenario of massive enough effect on the bottom line of the protocol or its users:

https://docs.sherlock.xyz/audits/watsons/judging

> Criteria for Issues: 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 more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

Here it is the attack on option holders from an outside attacker or LP, with insignificant costs (gas only) and massive enough effect on payouts. The probability of occurrence given the threshold and parameters is low, but it is typical for corner cases, which are usually identified during audits. Common case scenarios should be ideally caught by testing beforehand.

For the corner cases sake it matters to fix them as given big enough payoff the attacker can just setup a bot to track the right conditions. And it will not matter that it's 0.1% hypothetical probability for each case, maintaining lots of such bots will be quite profitable business at the expense of the corresponding protocol users.

hrishibhat

Escalation accepted

After considering all the comments/test cases & discussing internally although the possibility of this issue requires very specific scenarios to cause loss of funds, it still a valid issue. Upgrading to medium

Adding additional comment from the Watson:

> Just putting it to the extreme, if free liquidity is required to be 99% of all available, but Alice is the only depositor and wants to withdraw it will be allowed as the check happens before the withdraw, it's not simulating what happens after it. So she can pull the funds, and there will be a deficit with reduced board payout

sherlock-admin

> Escalation accepted
>
> After considering all the comments/test cases & discussing internally although the possibility of this issue requires very specific scenarios to cause loss of funds, it still a valid issue. Upgrading to medium
>
> Adding additional comment from the Watson:

> > Just putting it to the extreme, if free liquidity is required to be 99% of all available, but Alice is the only depositor and wants to withdraw it will be allowed as the check happens before the withdraw, it's not simulating what happens after it. So she can pull the funds, and there will be a deficit with reduced board payout

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

0xdomrom

> Just putting it to the extreme, if free liquidity is required to be 99% of all available, but Alice is the only depositor and wants to withdraw it will be allowed as the check happens before the withdraw, it's not simulating what happens after it. So she can pull the funds, and there will be a deficit with reduced board payout

Im more and more convinced the circuit breaker solves this issue outright. The free liquidity circuitbreaker check includes option debt, so the withdrawal will not go through and cause insolvency if that parameter is > 0, which it always should be.

From the test file itself:

```
// With default parameters:
// As free liquidity is < 1% of total NAV, circuit breaker is fired and
↪  withdrawal can't go through
{
  const snapshot = await takeSnapshot();
  await hre.f.c.liquidityPool.updateCBs();
  expect(await hre.f.c.liquidityPool.CBTimestamp()).gt(0);
  await restoreSnapshot(snapshot);
}
```

Maybe the recommendation should instead just ensure that the parameter cannot be == 0?

dmitriia

`liquidityCBThreshold` should definitely be positive and not too small.

Some comments:

- due to getHedgingLiquidity()'s `pendingDeltaLiquidity` approximate nature (#102) this number can be underestimated
- `optionValueDebt` used is a cached value (#74 and #99), so an attack can be conditional to catching the discrepancy of the actual vs cached value needed to settle:

SHERLOCK

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/OptionGreekCache.sol#L1019-L1022

```
/// @notice Get the current global net option value
function getGlobalOptionValue() external view returns (int) {
  return globalCache.netGreeks.netOptionValue;
}
```

- due to `call/putCollatScalingFactor` introduction freeing the collaterals impact is more limited, i.e. itself it can provide less than 1 `longScaleFactor`:

https://github.com/sherlock-audit/2022-12-lyra-dmitriia/blob/25f58622ae71e754ae3c220f4e8fdbec37864983/contracts/LiquidityPool.sol#L956-L970

```
uint availableQuote = totalQuote > usedQuote ? totalQuote - usedQuote : 0;

liquidity.pendingDeltaLiquidity = pendingDelta > availableQuote ? availableQuote
↪   : pendingDelta;
availableQuote -= liquidity.pendingDeltaLiquidity;

// Only reserve lockedColleratal x scalingFactor which unlocks more liquidity
// No longer need to lock one ETH worth of quote per call sold
uint reservedCollatLiquidity =
↪   lockedCollateral.quote.multiplyDecimal(lpParams.putCollatScalingFactor) +
  lockedCollateral.base.multiplyDecimal(basePrice).multiplyDecimal(lpParams.call⌋
↪   CollatScalingFactor);
liquidity.reservedCollatLiquidity = availableQuote > reservedCollatLiquidity
  ? reservedCollatLiquidity
  : availableQuote;

availableQuote -= liquidity.reservedCollatLiquidity;
liquidity.freeLiquidity = availableQuote > reservedTokenValue ? availableQuote -
↪   reservedTokenValue : 0;
```

I.e. this can be made possible by cache staleness (not necessary via keepers not being around, a sharp market movement can suffice) coupled with limiting the collateral impact on free liquidity. While the latter is design, for the former it might be recommended to add some rule for running keepers on volatility. Something like usual update period is 1 hour, shrinking to 1 minute with volatility growth, i.e. making period a RV formula to control the cache staleness.

jacksanford1

Based on @0xdomrom's comment above, Sherlock classifies this as "acknowledged" (but thought of as Low) by the protocol team.

jacksanford1

Comment from @0xdomrom on Discord:

I'm very confident having the free liquidity circuit breaker set to non 0 resolves the issue - and it would always be non 0. But given its a possibility with the parameter ranges I suppose it's a non-0 chance of occurring... sooo resolve as medium is fine

Issue M-3: GMXFuturesPoolHedger's getHedgingLiquidity mistreats the actual and theoretical liquidity

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/102

Found by

hyh

Summary

getHedgingLiquidity() uses theoretical liquidity as a measure of current liquidity conditions, which can be substantially wrong as market moves and hedging position accrues P&L.

Vulnerability Detail

There is an unaccounted difference between `pendingDeltaLiquidity` (how much is needed to bring theoretical is and theoretical to-be) and `_getCurrentHedgedNetDeltaWithSpot().multiplyDecimal(spotPrice) - _getAllPositionsValue(currentPositions).multiplyDecimal(futuresPoolHedgerParams.targetLeverage)` (theoretical and actual now).

This difference will emerge if current hedge be replaced with the target one, i.e. current position be closed and one with the target leverage and the current size opened instead. The difference is unrealized P&L and the leverage drift, both from P&L and say leverage parameter changes, if any, i.e. it's a cumulative drift from ideal conditions).

Impact

With free liquidity calculations being incorrect the hedging possibility control logic will not work as intended. One of the impacts is allowing for positions that cannot be subsequently hedged.

Without hedging the protocol is open to any delta originated losses, which can be massive and can have the net impact up to protocol insolvency.

There are no material prerequisites. Given the massive fund loss impact from the absence of the hedge setting the severity to be high.

Code Snippet

The difference between `pendingDeltaLiquidity` and `_getCurrentHedgedNetDeltaWithSpot().multiplyDecimal(spotPrice) -`

`_getAllPositionsValue(currentPositions).multiplyDecimal(futuresPoolHedgerParams.targetLeverage)`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L228-L250

```
/**
 * @notice Returns pending delta hedge liquidity and used delta hedge liquidity
 * @dev include funds potentially transferred to the contract
 * @return pendingDeltaLiquidity amount USD needed to hedge. outstanding order
↪  is NOT included
 * @return usedDeltaLiquidity amount USD already used to hedge. outstanding
↪  order is NOT included
 **/
function getHedgingLiquidity(
  uint spotPrice
) external view override returns (uint pendingDeltaLiquidity, uint
↪  usedDeltaLiquidity) {
  CurrentPositions memory currentPositions = _getPositions();

  usedDeltaLiquidity = _getAllPositionsValue(currentPositions);
  // pass in estimate spot price
  uint absCurrentHedgedDelta =
↪  Math.abs(_getCurrentHedgedNetDeltaWithSpot(currentPositions, spotPrice));
  uint absExpectedHedge = Math.abs(_getCappedExpectedHedge());

  if (absCurrentHedgedDelta > absExpectedHedge) {
    return (0, usedDeltaLiquidity);
  }

  pendingDeltaLiquidity = (absExpectedHedge -
↪  absCurrentHedgedDelta).multiplyDecimal(spotPrice).divideDecimal(
    futuresPoolHedgerParams.targetLeverage
  );
```

I.e. `pendingDelta` is not synchronized with the real liquidity situation, making liquidity's `pendingDeltaLiquidity` biased by the abovementioned difference:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L942-L959

```
function _getLiquidity(
  uint basePrice,
  uint totalPoolValue,
  uint reservedTokenValue,
  uint usedDelta,
  uint pendingDelta,
  uint longScaleFactor
```

```
) internal view returns (Liquidity memory) {
  Liquidity memory liquidity = Liquidity(0,0,0,0,0,0,0);
  liquidity.NAV = totalPoolValue;
  liquidity.usedDeltaLiquidity = usedDelta;

  uint usedQuote = totalOutstandingSettlements + totalQueuedDeposits;
  uint totalQuote =
↪   ConvertDecimals.convertTo18(quoteAsset.balanceOf(address(this)),
↪   quoteAsset.decimals());
  uint availableQuote = totalQuote > usedQuote ? totalQuote - usedQuote : 0;

  liquidity.pendingDeltaLiquidity = pendingDelta > availableQuote ?
↪   availableQuote : pendingDelta;
  availableQuote -= liquidity.pendingDeltaLiquidity;
```

This will also bias the linked amounts calculations, liquidity's `reservedCollatLiquidity`, `freeLiquidity` and `burnableLiquidity`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L958-L975

```
  liquidity.pendingDeltaLiquidity = pendingDelta > availableQuote ?
↪   availableQuote : pendingDelta;
  availableQuote -= liquidity.pendingDeltaLiquidity;

  // Only reserve lockedColleratal x scalingFactor which unlocks more liquidity
  // No longer need to lock one ETH worth of quote per call sold
  uint reservedCollatLiquidity =
↪   lockedCollateral.quote.multiplyDecimal(lpParams.putCollatScalingFactor) +
    lockedCollateral.base.multiplyDecimal(basePrice).multiplyDecimal(lpParams.ca ⌋
↪   llCollatScalingFactor);
  liquidity.reservedCollatLiquidity = availableQuote > reservedCollatLiquidity
    ? reservedCollatLiquidity
    : availableQuote;

  availableQuote -= liquidity.reservedCollatLiquidity;
  liquidity.freeLiquidity = availableQuote > reservedTokenValue ? availableQuote
↪   - reservedTokenValue : 0;
  liquidity.burnableLiquidity = availableQuote;
  liquidity.longScaleFactor = longScaleFactor;

  return liquidity;
}
```

This affects all the functions relying on liquidity to control protocol risk.

For example, the transferQuoteToHedge() limiting will be biased, sometimes limiting too strict, sometimes otherwise:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L1037-L1055

```
/**
 * @notice Sends quote to the PoolHedger.
 * @dev Transfer amount up to `pendingLiquidity + freeLiquidity`.
 * The hedger must determine what to do with the amount received.
 *
 * @param amount The amount requested by the PoolHedger.
 */
function transferQuoteToHedge(uint amount) external onlyPoolHedger returns
↪  (uint) {
  Liquidity memory liquidity = getLiquidity();

  uint available = liquidity.pendingDeltaLiquidity + liquidity.freeLiquidity;

  amount = amount > available ? available : amount;

  _transferQuote(address(poolHedger), amount);
  emit QuoteTransferredToPoolHedger(amount);

  return amount;
}
```

Also, new position opening is controlled with `trade.liquidity.freeLiquidity`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/OptionMarket.sol#L882-L916

```
/// @dev send/receive quote or base to/from LiquidityPool on position open
function _routeLPFundsOnOpen(TradeParameters memory trade, uint totalCost, uint
↪  feePortion) internal {
  if (trade.amount == 0) {
    return;
  }

  if (trade.optionType == OptionType.LONG_CALL) {
    liquidityPool.lockCallCollateral(trade.amount, trade.spotPrice,
↪   trade.liquidity.freeLiquidity);
    _transferFromQuote(msg.sender, address(liquidityPool), totalCost -
↪   feePortion);
    _transferFromQuote(msg.sender, address(this), feePortion);
  } else if (trade.optionType == OptionType.LONG_PUT) {
    liquidityPool.lockPutCollateral(trade.amount.multiplyDecimal(trade.strikePri ⌋
↪   ce), trade.liquidity.freeLiquidity);
    _transferFromQuote(msg.sender, address(liquidityPool), totalCost -
↪   feePortion);
    _transferFromQuote(msg.sender, address(this), feePortion);
  } else if (trade.optionType == OptionType.SHORT_CALL_BASE) {
    liquidityPool.sendShortPremium(
      msg.sender,
      trade.amount,
      totalCost,
      trade.liquidity.freeLiquidity,
      feePortion,
      true
    );
  } else {
    // OptionType.SHORT_CALL_QUOTE || OptionType.SHORT_PUT_QUOTE
    liquidityPool.sendShortPremium(
      address(shortCollateral),
      trade.amount,
      totalCost,
      trade.liquidity.freeLiquidity,
      feePortion,
      false
    );
  }
}
```

For example, lockCallCollateral() and lockPutCollateral():

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L586-L590

```
function lockCallCollateral(uint amount, uint spotPrice, uint freeLiquidity)
↪  external onlyOptionMarket {
  _checkCanHedge(amount, true);

  if (amount.multiplyDecimal(spotPrice).multiplyDecimal(lpParams.callCollatScali ⌋
↪  ngFactor) > freeLiquidity) {
    revert LockingMoreQuoteThanIsFree(
```

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L570-L572

```
function lockPutCollateral(uint amount, uint freeLiquidity) external
↪  onlyOptionMarket {
  if (amount.multiplyDecimal(lpParams.putCollatScalingFactor) > freeLiquidity) {
    revert LockingMoreQuoteThanIsFree(address(this), amount, freeLiquidity,
↪  lockedCollateral);
```

That is, if, for instance, there is a substantial enough unrealized negative P&L on the hedging position it will be recognized in NAV, but will not be properly accounted for in free liquidity value, so the positions will be allowed to be opened that will require hedging that may not be possible given real current liquidity, i.e. one after netting of the current P&L.

Increasing the leverage might not be always sufficient in such situations. When effective leverage be placed high enough this will bring the hedging to the brink of liquidation that will remove the hedging altogether and expose the pool.

Tool used

Manual Review

Recommendation

Consider using marked to market values in getHedgingLiquidity().

Discussion

hrishibhat

Sponsor comment:

> PoC or better recommendation would be nice, as it is quite hard to understand the writeup. - the pendingDeltaLiquidity value is just a rough estimation, it doesn't need to be super accurate. Adding in the extra complexity of unrealised PnL to that logic seems unncessary. - Would class this as low, medium might be fine

thec00n

Does not seem like this issue will be fixed (see sponsor comment).

dmitriia

Escalate for 70 USDC Not really understand the take of the comment. The issue is that due to not updating the current hedge to the market (i.e. not including current P&L to it) the immediate hedging possibility is evaluated incorrectly. This have the major impact of not being able to hedge, i.e. naked positions with losses up to insolvency, thus it was marked as high severity. Also, it shares the root cause, and due to the same nature and similar impact should have the same severity as #10. The root issue in both isn't that marking to market is not done like it's processed in current GMX implementation, it is that the update isn't performed.

sherlock-admin

> Escalate for 70 USDC Not really understand the take of the comment. The issue is that due to not updating the current hedge to the market (i.e. not including current P&L to it) the immediate hedging possibility is evaluated incorrectly. This have the major impact of not being able to hedge, i.e. naked positions with losses up to insolvency, thus it was

> marked as high severity. Also, it shares the root cause, and due to the same nature and similar impact should have the same severity as #10. The root issue in both isn't that marking to market is not done like it's processed in current GMX implementation, it is that the update isn't performed.

You've created a valid escalation for 70 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

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

0xdomrom

As the hedger can use locked collateral (i.e. collateral from selling naked calls and puts) there is more than enough collateral to cover all the excess that is required.

In the worst case of 0 being reserved ALL of the collateral can still be used for hedging. As such the estimation does not have to be accurate as per original comment.

Medium was probably too generous looking at the grading criteria. As this doesn't cause any real harm to LPs, I would class this as low.

dmitriia

Suppose nobody sold any options to AMM, and Alice, the only trader who interacted with the protocol, did two trades: at first she bought ETH ATM long call from AMM when ETH was 1200, say size is `100`, a hedge was open for `+0.5 * 100 = 50` delta, then ETH dropped to 1000 and Alice bough another ATM call with `100` size, the system hedged, say it is now `+0.5 * 100 + 0.4 * 100 = 90` delta and it's correct.

Now enters Bob who also wants to buy ETH long call from AMM of `1000` size. The decision whether to allow it should depend on the correct free liquidity estimation as the corresponding hedging should be opened (say hedge increased to `590`), but current `pendingDeltaLiquidity` estimate doesn't account for negative P&L sitting on the current hedge, so it underestimates the amount of liquidity that should be supplied to GMX to increase the position (let's say leverage there can't be increased anymore), so it allows Bob to open, while the `590` hedge can't be created as there is not enough LP funds for that.

0xdomrom

Yeah, I mean I agree theres an issue, but it is very minor

hrishibhat

Escalation rejected

Based on the discussions above & with the Sponsor, sherlock decided to keep the status of the issue unchanged.

sherlock-admin

> Escalation rejected
>
> Based on the discussions above & with the Sponsor, sherlock decided to keep the status of the issue unchanged.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.

jacksanford1

Based on @0xdomrom's comment above, Sherlock classifies this as "acknowledged" (but thought of as Low) by the protocol team.

Issue M-4: New SHORT_CALL_QUOTE positions hedging possibility is being checked incorrectly

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/100

Found by

hyh, Bahurum

Summary

New SHORT_CALL_QUOTE positions canHedge() check is performed as if pool delta is decreased as a result of a user selling call to it. As it is in fact increased, being a function of option type and not collateral type.

The check is inverted and can lead to blocking good trades, i.e. function unavailability, and not blocking of the bad ones as the check can be omitted in the result with `if !deltaIncreasing && expectedHedge >= 0 then allow` logic of GMXFuturesPoolHedger's canHedge(), i.e. allowing to open a position that cannot be hedged.

Vulnerability Detail

openPosition() with `SHORT_CALL_QUOTE` type invokes canHedge() check via sendShortPremium() with `deltaIncreasing == false`, while when call option is being sold to the pool delta increases no matter what collateral is being used, i.e. both for `SHORT_CALL_BASE` and `SHORT_CALL_QUOTE`, as the pool obtains a call with positive delta in both cases.

Impact

Core part of the impact is that `SHORT_CALL_QUOTE` trades aren't checked to be hedgeable.

Allowing a trade big enough so it cannot be hedged will force the protocol to be partially unhedged, i.e. sell some naked options. This can lead to delta initiated protocol wide losses quick enough.

As trade opening can't be protocol controlled in any way, being always initiated by a user, no low probability prerequisites looks to be needed in this case, so, given the possibility of substantial enough losses, setting the severity to be high.

Code Snippet

canHedge() treats `deltaIncreasing` as whether it `increases delta of the pool`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L334-L359

```
/**
 * @dev return whether a hedge should be performed
 */
function canHedge(uint /* amountOptions */, bool deltaIncreasing) external view
↪  override returns (bool) {
  ...

  if (Math.abs(expectedHedge) <= Math.abs(currentHedge)) {
    // Delta is shrinking (potentially flipping, but still smaller than current
↪   hedge), so we skip the check
    return true;
  }

  if (deltaIncreasing && expectedHedge <= 0) {
    // expected hedge is negative, and trade increases delta of the pool
    return true;
  }

  if (!deltaIncreasing && expectedHedge >= 0) {
    return true;
  }
```

It's invoked via _checkCanHedge():

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L1028-L1035

```
function _checkCanHedge(uint amountOptions, bool increasesDelta) internal view {
  if (address(poolHedger) == address(0)) {
    return;
  }
  if (!poolHedger.canHedge(amountOptions, increasesDelta)) {
    revert UnableToHedgeDelta(address(this), amountOptions, increasesDelta);
  }
}
```

sendShortPremium() uses _checkCanHedge() with `deltaIncreasing = increasesDelta = isCall`:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/LiquidityPool.sol#L635-L660

```
/**
 * @notice Sends premium user selling an option to the pool.
 * @dev The caller must be the OptionMarket.
 *
 * @param recipient The address of the recipient.
 * @param amountContracts The number of contracts sold to AMM.
 * @param premium The amount to transfer to the user.
 * @param freeLiquidity The amount of free collateral liquidity.
 * @param reservedFee The amount collected by the OptionMarket.
 */
function sendShortPremium(
  address recipient,
  uint amountContracts,
  uint premium,
  uint freeLiquidity,
  uint reservedFee,
  bool isCall
) external onlyOptionMarket {
  if (premium + reservedFee > freeLiquidity) {
    revert SendPremiumNotEnoughCollateral(address(this), premium, reservedFee,
↪   freeLiquidity);
  }

  // only blocks opening new positions if cannot hedge
  _checkCanHedge(amountContracts, isCall);
  _sendPremium(recipient, premium, reservedFee);
}
```

_routeLPFundsOnOpen() calls liquidityPool.sendShortPremium() with `isCall == false` for `SHORT_CALL_QUOTE` type:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/OptionMarket.sol#L896-L915

```
} else if (trade.optionType == OptionType.SHORT_CALL_BASE) {
  liquidityPool.sendShortPremium(
    msg.sender,
    trade.amount,
    totalCost,
    trade.liquidity.freeLiquidity,
    feePortion,
    true
  );
} else {
  // OptionType.SHORT_CALL_QUOTE || OptionType.SHORT_PUT_QUOTE
  liquidityPool.sendShortPremium(
    address(shortCollateral),
    trade.amount,
    totalCost,
    trade.liquidity.freeLiquidity,
    feePortion,
    false
  );
}
```

_openPosition() calls _routeLPFundsOnOpen():

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/OptionMarket.sol#L560-L595

```
/**
 * @dev Opens a position, which may be long call, long put, short call or short
↪  put.
 */
function _openPosition(TradeInputParameters memory params) internal returns
↪  (Result memory result) {
  (TradeParameters memory trade, Strike storage strike, OptionBoard storage
↪  board) = _composeTrade(
    ...
  );
  OptionMarketPricer.TradeResult[] memory tradeResults;
  (trade.amount, result.totalCost, result.totalFee, tradeResults) = _doTrade(
    ...
  );

  int pendingCollateral;
  // collateral logic happens within optionToken
  (result.positionId, pendingCollateral) = optionToken.adjustPosition(
    ...
  );

  uint reservedFee =
↪  result.totalFee.multiplyDecimal(optionMarketParams.feePortionReserved);

  _routeLPFundsOnOpen(trade, result.totalCost, reservedFee);
```

_openPosition() is called by user-facing openPosition():

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/OptionMarket.sol#L505-L514

```
/**
 * @notice Attempts to open positions within cost bounds.
 * @dev If a positionId is specified that position is adjusted accordingly
 *
 * @param params The parameters for the requested trade
 */
function openPosition(TradeInputParameters memory params) external nonReentrant
↪  returns (Result memory result) {
  result = _openPosition(params);
  _checkCostInBounds(result.totalCost, params.minTotalCost, params.maxTotalCost);
}
```

This way for `SHORT_CALL_QUOTE` type sendShortPremium() involves canHedge() check with `deltaIncreasing == false`, i.e. as if `increases delta of the pool` is false:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/OptionMarket.sol#L906-L914

```
// OptionType.SHORT_CALL_QUOTE || OptionType.SHORT_PUT_QUOTE
liquidityPool.sendShortPremium(
  address(shortCollateral),
  trade.amount,
  totalCost,
  trade.liquidity.freeLiquidity,
  feePortion,
  false
);
```

In the same time `SHORT_CALL_QUOTE` is selling call to the pool for a premium, an operation that increases delta of the pool. I.e. no matter what collateral is used, the pool has additional call option being sold to it and this way has its delta increased. I.e. the hedging check in this case is reverted and can block good trades or allow the ones that can't be hedged.

That is, `remaining < absHedgeDiff.multiplyDecimal(futuresPoolHedgerParams.marketDepthBuffer)` can hold, meaning that there is a shortage, but the canHedge() was already returned `true` via `!deltaIncreasing && expectedHedge >= 0` condition, having `deltaIncreasing == false` for this type of call:

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L337-L373

```
function canHedge(uint /* amountOptions */, bool deltaIncreasing) external view
↪  override returns (bool) {
  ...

  if (deltaIncreasing && expectedHedge <= 0) {
    // expected hedge is negative, and trade increases delta of the pool
    return true;
  }

  if (!deltaIncreasing && expectedHedge >= 0) {
    return true;
  }
  // remaining is the number of us dollars that can be hedged
  uint remaining = ConvertDecimals.convertTo18(
    (vault.poolAmounts(address(baseAsset)) -
↪   vault.reservedAmounts(address(baseAsset))),
    baseAsset.decimals()
  );

  uint absHedgeDiff = (Math.abs(expectedHedge) - Math.abs(currentHedge));
  if (remaining <
↪   absHedgeDiff.multiplyDecimal(futuresPoolHedgerParams.marketDepthBuffer)) {
    return false;
  }

  ...
}
```

Tool used

Manual Review

Recommendation

Consider checking hedging possibility for `SHORT_CALL_QUOTE` trades with `_checkCanHedge(amountContracts, true)`.

Discussion

hrishibhat

Sponsor comment:

> this is valid as it is about incorrect input to the canHedge function. I think it's a duplicate of 86

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/751782f91d817408bbb1ab29e9bf530823bd95dd

hrishibhat

Considering the issue is possible for a SHORT_CALL_QUOTE trade, with additional sponsor comments:

> there's increased delta risk that that is unexpected for LPs. That can lead to both positive or negative gains potentially.

considering this issue as a valid medium

thec00n

Fixes incorrect input to the `canHedge()` function in `LiquidityPool`. Fix LGTM

Issue M-5: Poor validation or prices reported by Chainlink oracles

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/90

Found by

TrungOre, Bahurum, Jeiwan, hansfriese, GalloDaSballo, ctf_sec

Summary

The `GMXAdapter` contract integrates with Chainlink feeds, however prices reported by Chainlink feeds are not thoroughly validated. This exposes the protocol to the negative impact of extreme market events, possible malicious activity of Chainlink data feeders or its contracts, potential delays, and outages.

Vulnerability Detail

The `_getChainlinkPrice` function of `GMXAdapter` reads a price from a Chainlink feed and ignores the `updatedAt` value returned by `latestRoundData` (GMXAdapter.sol#L192):

```
(, int answer, , , ) = assetPriceFeed.latestRoundData();
if (answer <= 0) {
  revert InvalidAnswer(address(this), answer);
}
```

Besides the `answer`, `latestRoundData` also returns `updatedAt`, the timestamp of when the round was updated. Chainlink Price Feeds do not provide streaming data. Rather, the aggregator updates its `latestAnswer` when the value deviates beyond a specified threshold or when the heartbeat idle time has passed. Due to outages or malicious activity of Chainlink data feeders, prices may be delayed or even become stale.

`latestRoundData` also returns `roundID` and `answeredInRound` values, which can be used to check for price reporting round completeness. However, the values are ignored and not validated, which exposes the protocol to any Chainlink issues with completing price reporting rounds.

Also, the `answer` is not checked against reasonable price limits: while there's a check for a zero or negative price, an invalid price may still be above zero.

Impact

By not checking the `updateAt` value, `GMXAdapter` becomes exposed to outages in Chainlink or malicious activity of its data feeders, which may lead to incorrect pricing during liquidations or any other activity that uses the `PriceType.REFERENCE` price type.

In case a Chainlink price feed has failed to finalize a round, `GMXAdapter` won't detect a stale or an invalid price, which will affect all operations that use the reference price type.

In case a price that's outside of the reasonable limits of an asset is reported (due to an outage in Chainlink or malicious activity), `GMXAdapter` will fail to detect such price. As a result, any operation that uses the reference price type will be impaired.

Code Snippet

GMXAdapter.sol#L192-L195:

```
(, int answer, , , ) = assetPriceFeed.latestRoundData();
if (answer <= 0) {
  revert InvalidAnswer(address(this), answer);
}
```

Tool used

Manual Review

Recommendation

Consider following the Monitoring data feeds recommendations from Chainlink and:

1. start checking the `updatedAt` value returned by `latestRoundData`: ensure it doesn't fall outside of the hearbeat period of the price feed;
2. start checking the `roundID` and `answeredInRound` values: ensure `answeredInRound` is always greater or equal to `roundID` (See for more details);
3. start reading the `maxAnswer` and `minAnswer` from the Chainlink aggregator and ensure that prices reported by `latestRoundData` are always within the reasonable limit;
4. however, be aware that the `maxAnswer` and `minAnswer` values are not immutable and they may change when the Chainlink aggregator is changed.

Discussion

hrishibhat

Sponsor comment:

> Duplicate of many that have mentioned CL staleness - Recommentation to "ensure `answeredInRound` is always greater or equal to `roundID`" would always revert

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/4b931d59d03b86e13e1fa7eefc5a2abdcbd2b2a7

thec00n

Update https://github.com/lyra-finance/lyra-protocol/pull/7/commits/7262e3ea8d4e5bcebf05d4fe16d0917e67e7dd28

Checks `updatedAt` from `latestRoundData()` and performs a staleness check.

The other recommendations are not implemented.

jacksanford1

Due to a subset of the recommendations being implemented, Sherlock classifies this as partially fixed and partially acknowledged by the protocol team.

jacksanford1

From @0xdomrom on Discord:

> yep unnecessary for our use case from my understanding/our team's understanding

Issue M-6: Lack of claim method for referral fee in GMX-FuturesPoolHedger

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/73

Found by

ctf_sec

Summary

The referral fee airdropped to GMXFuturesPoolHedger will be lost.

Vulnerability Detail

The default referral fee code in GMXFuturesPoolHedger is set

```
bytes32 public referralCode = bytes32("LYRA");
```

This referral code is used when used to create increase position position request.

```
uint executionFee = _getExecutionFee();
bytes32 key = positionRouter.createIncreasePosition{value: executionFee}(
  path,
  address(baseAsset), // index token
  collateralDelta, // amount in via router is in the native currency decimals
  0, // min out
  _convertToGMXPrecision(sizeDelta),
  isLong,
  acceptableSpot,
  executionFee,
  referralCode,
  address(0)
);
```

According to GMX documentation:

https://gmxio.gitbook.io/gmx/referrals#tiers

If using referral code, the discount is applied

```
Tier 1: 5% discount for traders, 5% rebates to referrer
Tier 2: 10% discount for traders, 10% rebates to referrer
Tier 3: 10% discount for traders, 15% rebates to referrer paid in ETH / AVAX, 5%
↪  rebates to referrer paid in esGMX
```

> Rebates and discounts apply on the opening and closing fees for leverage trading.
>
> The opening and closing fees are 0.1% on GMX, there is no price impact for trades and zero spread for tokens like BTC and ETH, rebates are calculated before user discounts so referrers earn on the full maker fee and from what would otherwise be spread on other exchanges. As a result, referrers would earn equivalent amounts of rebates per volume on GMX when compared to other referral programs.
>
> Note that there is a cap of 5000 esGMX distributed per week. If the price of GMX is $30 the full 5% bonus can be paid for total Tier 3 referral volumes up to $3 billion per week. esGMX tokens distributed for this program will not require GMX or GLP to vest, the vault to vest the tokens will be available towards the end of Q1 2023.
>
> The price of esGMX will be based on the 7 day TWAP of GMX. Wallet providers and other protocols will be eligible for Tier 2 and Tier 3 rewards as well.

The trader fee discount that is airdropped to the GMXFuturesPoolHedger contract is lost because there is no such method in GMXFuturesPoolHedger to locked ERC20 token.

Impact

The trader fee discount that is airdropped to the GMXFuturesPoolHedger contract is lost because there is no such method in GMXFuturesPoolHedger to locked ERC20 token.

Code Snippet

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L95

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L157-L161

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L670-L684

Tool used

Manual Review

Recommendation

We recommend the project add a admin function to claim stucked ERC20 token including the claiming the airdropped GMX trading fee refund.

Discussion

hrishibhat

Sponsor comment:

> Duplicate of another that has mentioned GMX trading rebates - waiting on their team response

Sponsor comment on a similar issue #58 :

> Invalid referrals aren't taken out of the fee at the time of the trade, they are returned asyncronously afterwards

hrishibhat

Sponsor comment:

> have heard back from the GMX team, and yeah it's invalid as the fee rebates are airdropped periodically rather than part of the transaction

hrishibhat

@0xdomrom would these be airdropped periodically to a different address? Or the GMXFuturesPoolHedger contract? In the second case, I think Watson here is referring to GMXFuturesPoolHedger's inability to withdraw airdropped tokens.

0xdomrom

@hrishibhat

~~They would be airdropped to the GMXFuturesPoolHedger contract as quoteAsset or baseAsset.~~

~~There is a function to withdraw them, sendAllFundsToLP(). Even if they are in baseAsset they would be transferred to the LP and then converted back to quoteAsset with no issue.~~

~~Only referrer rebates are in other tokens. The hedger contract would not be a referrer.~~

~~Will triple check with the GMX team to make sure, but from the documentation there and what they've said that seems to be the case.~~

Turns out all rebates are sent in as wETH... So perfectly fine for the ETH market, but any other market wouldn't be able to handle them. Will add in a fix for this. Medium as severity seems appropriate.

There are rare cases where excess might be sent in as esGMX, but a general catch all to return arbitrary funds callable by the owner seems fine to handle that case.

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/83de4d893a61e6728ed776763e66a1c1d99ce8a9

thec00n

Updates https://github.com/lyra-finance/lyra-protocol/pull/7/commits/7262e3ea8d4e5bcebf05d4fe16d0917e67e7dd28#diff-0629a6b4a172d8cce8c9c35fba9ee9111acfdc203c0a2e7bec2062ec3dd7a7df

Added `sendAllFundsToLP()` to recover base, quote and WETH and send back to `LiquidityPool`. Also added `recoverFunds()` to extract none base/quote/weth tokens. Fix LGTM

Issue M-7: GMXFuturePoolHedger does not implement receive fallback function to receive the execution fee refund when increase position and decrease position are canceled

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/49

Found by

ctf_sec

Summary

GMXFuturePoolHedger does not implement receive fallback function to receive the execution fee refund when increase position and decrease position are canceled

Vulnerability Detail

When incresae or decrease position, the execution fee is sent to the PositionRouter contract.

```
uint executionFee = _getExecutionFee();
bytes32 key = positionRouter.createIncreasePosition{value: executionFee}(
  path,
  address(baseAsset), // index token
  collateralDelta, // amount in via router is in the native currency decimals
  0, // min out
  _convertToGMXPrecision(sizeDelta),
  isLong,
  acceptableSpot,
  executionFee,
  referralCode,
  address(0)
);
```

and

```
uint executionFee = _getExecutionFee();
bytes32 key = positionRouter.createDecreasePosition{value: executionFee}(
  path,
  address(baseAsset),
  // CollateralDelta for decreases is in PRICE_PRECISION rather than asset
↪  decimals like for opens...
  // In the case of closes, 0 must be passed in
  isClose ? 0 : _convertToGMXPrecision(collateralDelta),
  _convertToGMXPrecision(sizeDelta),
  isLong,
  address(liquidityPool),
  acceptableSpot,
  0,
  executionFee,
  false,
  address(0)
);
```

According to the GMX doc,

https://gmx-io.notion.site/gmx-io/GMX-Technical-Overview-47fc5ed832e243afb9e97e8a4a036353

The PositionRouter contract handles a two part transaction process for increasing or decreasing long / short positions, this process helps to reduce front-running issues:

1. A user sends the request to increase / decrease a position to the PositionRouter
2. A keeper requests the index price from an aggregate of exchanges
3. The keeper then executes the position at the current index price
4. If the position cannot be executed within the allowed slippage the request is cancelled and the funds are sent back to the user

note the case 4:

If the position cannot be executed within the allowed slippage the request is cancelled and the funds are sent back to the user

When the request is canceled, the execution fee is supposed to be refunded to the GMXFuturePoolHedger

```
/**
 * @dev cancel outstanding order in case the GMX keeper bot is not working
↪  properly.
 */
function cancelPendingOrder() external nonReentrant {
  if (lastOrderTimestamp + futuresPoolHedgerParams.minCancelDelay >
↪  block.timestamp)
    revert CancellationDelayNotPassed();

  if (_hasPendingIncrease()) {
    bool success = positionRouter.cancelIncreasePosition(pendingOrderKey,
↪  address(this));
    emit OrderCanceled(pendingOrderKey, success);
    }
    if (_hasPendingDecrease()) {
      bool success = positionRouter.cancelDecreasePosition(pendingOrderKey,
 ↪    address(this));
      emit OrderCanceled(pendingOrderKey, success);
    }

    pendingOrderKey = bytes32(0);
  }
```

note the function call cancalIncreasePosition and cancelDecreasePosition

the code above pass in address(this) as the executionFeeReceiver

Now we look into the code on GMX side

https://github.com/gmx-io/gmx-contracts/blob/d5ad12288a79c672ad7553bd772c09bfca231c11/contracts/core/PositionRouter.sol#L459

```
function cancelIncreasePosition(bytes32 _key, address payable
 ↪  _executionFeeReceiver) public nonReentrant returns (bool) {
    IncreasePositionRequest memory request = increasePositionRequests[_key];
    // if the request was already executed or cancelled, return true so that the
 ↪  executeIncreasePositions loop will continue executing the next request
    if (request.account == address(0)) { return true; }

    bool shouldCancel = _validateCancellation(request.blockNumber,
 ↪  request.blockTime, request.account);
    if (!shouldCancel) { return false; }

    delete increasePositionRequests[_key];

    if (request.hasCollateralInETH) {
        _transferOutETHWithGasLimitIgnoreFail(request.amountIn,
 ↪  payable(request.account));
    } else {
        IERC20(request.path[0]).safeTransfer(request.account, request.amountIn);
    }

   _transferOutETHWithGasLimitIgnoreFail(request.executionFee,
 ↪  _executionFeeReceiver);

    emit CancelIncreasePosition(
        request.account,
        request.path,
        request.indexToken,
        request.amountIn,
        request.minOut,
        request.sizeDelta,
        request.isLong,
        request.acceptablePrice,
        request.executionFee,
        block.number.sub(request.blockNumber),
        block.timestamp.sub(request.blockTime)
    );

    _callRequestCallback(request.callbackTarget, _key, false, true);

    return true;
}
```

note the function call

```
_transferOutETHWithGasLimitIgnoreFail(request.executionFee,
↪   _executionFeeReceiver);
```

Which calls:

https://github.com/gmx-io/gmx-contracts/blob/d5ad12288a79c672ad7553bd772c09bfca231c11/contracts/core/BasePositionManager.sol#L279

```
function _transferOutETHWithGasLimitIgnoreFail(uint256 _amountOut, address
↪   payable _receiver) internal {
    IWETH(weth).withdraw(_amountOut);

    // use `send` instead of `transfer` to not revert whole transaction in case
↪   ETH transfer was failed
    // it has limit of 2300 gas
    // this is to avoid front-running
    _receiver.send(_amountOut);
}
```

As the comment suggest, on GMX side, _receiver.send is used intentionally to make sure the failure on execution fee refunding will be block the cancel.

Then cancel request from cancelPendingOrdel in GMXFuturePoolHedger may go through, but because the GMXFuturePoolHedger does not implement the receive fallback function, the execution fee refund fail sliently and the executee fee is lost even though the request is canceled.

Impact

the execution fee refund fail sliently and the executee fee is lost even though the request is canceled.

Code Snippet

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L417-L436

Tool used

Manual Review

Recommendation

We recommend the project implement receive fallback in GMXFuturePoolHedger to make sure the execution fee is received properly.

```
receive() external payable {

}
```

Discussion

hrishibhat

Sponsor comment:

> low or medium Need to verify the validity, was under the impression funds would be returned in baseAsset, but that could be incorrect.

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/aa301dee950ecf35f8253332f0121ed21bb4dad9

thec00n

A receive fallback function was added to `GMXFuturePoolHedger`. LGTM

Issue M-8: No check for active Arbitrum and optimistic Sequencer in WSTETH Oracle

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/47

Found by

Bahurum, ctf_sec

Summary

The protocol wants to deploy for Arbitrum and optimism

Chainlink recommends that all Optimistic L2 oracles consult the Sequencer Uptime Feed to ensure that the sequencer is live before trusting the data returned by the oracle. There is no such check implemented in the GMXAdapter.sol

Vulnerability Detail

If the Arbitrum Sequencer goes down, oracle data will not be kept up to date, and thus could become stale. However, users are able to continue to interact with the protocol directly through the L1 optimistic rollup contract. You can review Chainlink docs on L2 Sequencer Uptime Feeds for more details on this.

As a result, users may be able to use the protocol while oracle feeds are stale. Then outdated price can be used to settle trade.

Impact

If the Arbitrum or optimism sequencer goes down, the protocol will allow users to continue to operate at the previous (stale) rates.

Code Snippet

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXAdapter.sol#L179-L198

Tool used

Manual Review

Recommendation

We recommend add the sequencer active check

```
function isSequencerActive() internal view returns (bool) {
    (, int256 answer, uint256 startedAt,,) = sequencer.latestRoundData();
    if (block.timestamp - startedAt <= GRACE_PERIOD_TIME || answer == 1)
        return false;
    return true;
}
```

and

```
if (!isSequencerActive()) revert Errors.L2SequencerUnavailable();
```

Discussion

hrishibhat

Sponsor comment:

> Valid, similar to 041 but slightly different A staleness check will be added for the chainlink price. The contract will revert if the feed hasn't been updated for too long.

thec00n

Latest commit https://github.com/lyra-finance/lyra-protocol/pull/7/commits/7262e3ea8d4e5bcebf05d4fe16d0917e67e7dd28 or any previous commits do not fix this issue.

jacksanford1

Classified as acknowledged by the protocol team. @0xdomrom's comment from Discord:

> same issue as 90, won't be added as it isn't really relevant to our use case from my understanding

Issue M-9: `canHedge()` does not check short exposure nor is the current option amount included in `absHedgeDiff`

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/14

Found by

thec00n

Summary

The `GMXFuturesPoolHedger` contract provides a utility function, `canHedge()`, used by the `LiquidityPool` to determine if option issuance should be performed. The function misses checks that should block option issuance under extreme conditions.

Vulnerability Detail

The function `canHedge()` queries the GMX `Vault` remaining base asset amount that can be used for hedging on line 362 and then calculates `absHedgeDiff`, which is the difference between the current hedge and the expected hedge. A check is then performed to ensure that the `Vault` has sufficient market depth for hedging long on line 368 compared to `absHedgeDiff`. It does not perform the check for the short side.

The current amount from an option sale is not considered in`canHedge()` and the variable `amountOptions` on line 337 is not used. `absHedgeDiff` does not consider the option amount when it performs the market depth check with the `Vault`'s remaining.

Impact

The `LiquidityPool` could sell options under extreme market conditions and when it should block option issuance.

Code Snippet

```
function canHedge(uint /* amountOptions */, bool deltaIncreasing) external view
↪  override returns (bool) {
  if (!futuresPoolHedgerParams.vaultLiquidityCheckEnabled) {
    return true;
  }

  uint spotPrice = _getSpotPrice();
  CurrentPositions memory positions = _getPositions();
  int expectedHedge = _getCappedExpectedHedge();
  int currentHedge = _getCurrentHedgedNetDeltaWithSpot(positions, spotPrice);

  if (Math.abs(expectedHedge) <= Math.abs(currentHedge)) {
    // Delta is shrinking (potentially flipping, but still smaller than current
↪   hedge), so we skip the check
    return true;
  }

  if (deltaIncreasing && expectedHedge <= 0) {
    // expected hedge is negative, and trade increases delta of the pool
    return true;
  }

  if (!deltaIncreasing && expectedHedge >= 0) {
    return true;
  }

  // remaining is the number of us dollars that can be hedged
  uint remaining = ConvertDecimals.convertTo18(
    (vault.poolAmounts(address(baseAsset)) -
↪   vault.reservedAmounts(address(baseAsset))),
    baseAsset.decimals()
  );

  uint absHedgeDiff = (Math.abs(expectedHedge) - Math.abs(currentHedge));
  if (remaining <
↪   absHedgeDiff.multiplyDecimal(futuresPoolHedgerParams.marketDepthBuffer)) {
    return false;
  }

  return true;
}
```

https://github.com/sherlock-audit/2022-12-lyra/blob/main/contracts/GMXFuturesPoolHedger.sol#L337-L372

Tool used

Manual Review

Recommendation

Perform an additional check for the short side and fetch the remaining quote from GMX's `Vault` and compare it with `absHedgeDiff`.

Consider the current option amount when performing the market depth check with GMX's `Vault` remaining.

Discussion

hrishibhat

Sponsor comment:

> Valid

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/6f0ee00187ea91eb92dcfa090fb6c398b8b51822

thec00n

`canHedge()` does additional check for the short side and fetches the remaining quote from GMX's `Vault` and compare it with `absHedgeDiff`. LGTM

Issue M-10: Swap fees are not applied to `collateralDelta`

Source: https://github.com/sherlock-audit/2022-12-lyra-judging/issues/11

Found by

thec00n

Summary

Missing inclusion of swap fees can lead to inaccuracies in hedging when adjusting long positions on GMX.

Vulnerability Detail

`_collectSwapFees()` is used to calculate the swap fee and deduct it from the token output amount in the GMX `Vault` contract on line 545. `GMXFuturesPoolHedger` does not include the swap fee when it calculates the `collateralDelta` for long positions in `_decreasePosition()` and `_increasePosition()`.

Impact

`GMXFuturesPoolHedger` does not accurately adjust the hedge position.

Code Snippet

```
    uint256 feeBasisPoints = vaultUtils.getSwapFeeBasisPoints(_tokenIn,
 ↪  _tokenOut, usdgAmount);
    uint256 amountOutAfterFees = _collectSwapFees(_tokenOut, amountOut,
 ↪  feeBasisPoints);

    _increaseUsdgAmount(_tokenIn, usdgAmount);
    _decreaseUsdgAmount(_tokenOut, usdgAmount);

    _increasePoolAmount(_tokenIn, amountIn);
    _decreasePoolAmount(_tokenOut, amountOut);

    _validateBufferAmount(_tokenOut);

    _transferOut(_tokenOut, amountOutAfterFees, _receiver);

    emit Swap(_receiver, _tokenIn, _tokenOut, amountIn, amountOut,
 ↪  amountOutAfterFees, feeBasisPoints);

    useSwapPricing = false;
    return amountOutAfterFees;
}
```

https://github.com/gmx-io/gmx-contracts/blob/master/contracts/core/Vault.sol#L544-L561

Tool used

Manual Review

Recommendation

Swap fees should be applied to `collateralDelta` before `createIncreasePosition()` or `createDecreasePosition()` is called on GMX's `PositionRouter`.

Discussion

hrishibhat

Sponsor comment:

> Valid, but seems only relevant to increasePosition from testing

hrishibhat

Fix: https://github.com/lyra-finance/lyra-protocol/pull/7/commits/6f0ee00187ea91eb92dcfa090fb6c398b8b51822

thec00n

Fix update https://github.com/lyra-finance/lyra-protocol/pull/7/commits/cdcc2252f9a444e125b2725a8603fc3e70999371

thec00n

`getSwapFeeBP()` calculates swap fee and adds it to `collateralDelta` when positions are increased. Lyra tested and confirmed that fees do not need to be added when decreases positions. LGTM.