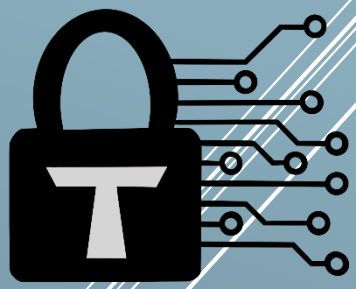
Trust Security

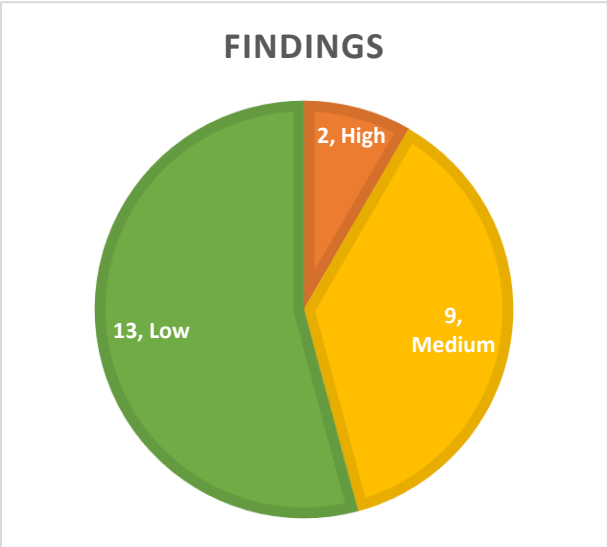


Smart Contract Audit

Rysk Beyond

09/03/23

Executive summary

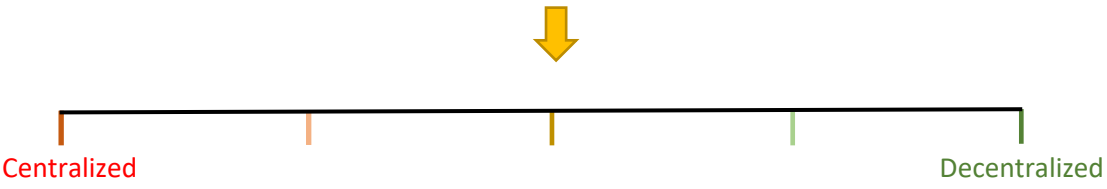


Category	Options
Audited file count	20
Lines of Code	4511
Auditor	Trust
Auditor	100proof
Time period	20/02-09/03

Findings

Severity	Total	Open	Fixed	Acknowledged
High	2	-	1	1
Medium	9	-	8	1
Low	13	-	7	6

Centralization score



Signature

Digitally signed by Trust
Date: 2023-03-29 17:54:39

Document properties

Versioning

Version	Date	Description
0.1	09/03/23	Client report
0.2	29/03/23	Mitigation review
0.3	29/03/23	Mitigation review #2

Contact

Or Cyngiser, AKA Trust

boss@trustindistrust.com

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate. Specifically, the object of focus has been migration from Rysk Alpha to Rysk Beyond and its possible repercussions.

Scope

- libraries/AccessControl.sol
- libraries/BlackScholes.sol
- libraries/CombinedActions.sol
- libraries/CustomErrors.sol
- libraries/NormalDist.sol
- libraries/OptionsCompute.sol
- libraries/OpynInteractions.sol
- libraries/RyskActions.sol
- libraries/SABR.sol
- libraries/Types.sol
- Accounting.sol
- AlphaPortfolioValuesFeed.sol
- BeyondPricer.sol
- LiquidityPool.sol
- OptionExchange.sol
- OptionCatalogue.sol
- OptionRegistry.sol
- PriceFeed.sol
- VolatilityFeed.sol
- hedging/GMXHedgingReactor.sol

Repository details

- **Repository URL:** https://github.com/rysk-finance/dynamic-hedging
- **Commit hash:** 541df6f606f09ab690af270e636c0f4bdb1f6bca
- **Mitigation review hash:** 28a36d4f768aef6194005ad37f35b06e7b4d95d6
- **Mitigation review 2 hash:** 1b0f75cc529545d52710b64c16d9a94983620d26

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	The Project has structured the codebase to cope well with the inherent complexity involved.
Documentation	**Excellent**	Project is very well documented.
Best practices	**Good**	Project mostly adheres to industry standards.
Centralization risks	**Moderate**	The protocol cannot be considered completely decentralized with the way pricing and adjustments are handled by the team. However this is still a big step in the right direction.

Findings

High severity findings

TRST-H-1 Unbounded slippage in OptionExchange::_buyOption() and _sellOption() makes users vulnerable to sandwich attack

- **Category:** MEV
- **Source:** OptionExchange.sol
- **Status:** Fixed

Description

Rysk prices options in a way that incentivizes reduced vault exposure to the underlying. It discounts trades reducing exposure and upsizes fees when trades increase exposure, which is performed in *_getSlippageMultiplier*() in BeyondPricer. For example, an increased delta of +x when the vault is at +10x will price a much higher premium than when vault is at +x. Meanwhile, when users purchase or sell options via the OptionExchange, they do not specify a minimum or maximum execution price. This combination of factors opens up the following sandwich attack when user calls *buyOption()*:

1) Attacker calls *buyOption()* with a large oToken request, driving down the net exposure of the vault to a very large negative value.
2) Victim's TX is sandwiched, executing the *buyOption()* when short exposure is already high and causing a very high premium to be charged.
3) Attacker calls *sellOption()* and sells the tokens from (1). They return the vault back to previous exposure level except victim's TX, picking up the high premium paid by the victim.

Since users of the protocol don't know the quote ahead of time, it is likely they will perform a max approval to OptionExchange which would allow the attacker to drain essentially their entire wallet, given a large enough initial bankroll to fund the manipulation. Additionally, a symmetrical attack can be performed on a *sellOption()* transfer, where the premium the protocol will pay for the user's oTokens can be lowered to a negligible value with the opposite manipulation.

Recommended mitigation

Incorporate slippage control values in the user-supplied trade structure, and disallow trades that cause slippage to exceed the specified amount.

Team response

Resolved.

Mitigation review

OptionExchange receives the max slippage user is willing to undertake, and correctly validates that the dynamic premium does not exceed it.

TRST-H-2 Decreasing positions in GMXHeadingReactor may lead to unhealthy positions

- **Category:** Logical Flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Acknowledged

Description

When trying to decrease the existing GMX position in *_changePosition()*, reached from *hedgeDelta()*, it uses *_getCollateralSizeDeltaUsd()* to calculate the changed delta. When the position is losing, the function takes into account the unrealized losses and deducts them from the collateral to decrease, possibly making the amount go negative (highlighted in **bold**).

```
uint256 adjustedAmount = _amount < position[0].div(position[2])
    ? _amount
    : position[0].div(position[2]);
uint256 d = adjustedAmount.mul(position[2]).div(position[0]);
{
    // we need to adjust the collateral to remove by 1% to account for
oracle price changes between this call and the gmx callback
    collateralToRemove =
        (((int256(position[1] / 1e12) -
            ((int256(position[0]) / 1e12).mul(1e18 -
int256(d)).div(int256(leverageFactor)))) -
            int256(position[8] / 1e12)) * collateralRemovalPercentage) /
        10000;
}
```

Later, if the result is negative, it simply changes it to 0.

```
if (collateralToRemove < 0) {
    adjustedCollateralToRemove = uint256(0);
} else {
    adjustedCollateralToRemove = uint256(collateralToRemove);
```

This is an issue because instead of actually increasing the amount of collateral required, the function only does not remove any existing collateral. As losses are realized, it may lead to a borderline position, which is close to liquidation. The position's health can eventually be replenished by a future *update()* call, however by that time it may already be too late and the position may be liquidated.

Recommended mitigation

Rewrite the code to first increase the collateral by the required amount, and only then decrease the position size and realize losses.

Team response

Acknowledged. Health is verified by an off-chain bot setup so the increased complexity in the fix suggested is not necessary.

Medium severity findings

TRST-M-1 GmxHedgingReactor::getPoolDenominatedValue() does not include pending increase of position

- **Category:** Logical flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

GMX has a two-step process for increasing/decreasing positions where a request is submitted and remains in a pending state until it is executed by a keeper. If *getPoolDenominatedValue()* is called just after a call to GMX's *createIncreasePosition()* (but before the request is executed), it will not take the increase into account leading to incorrect NAV evaluation in *executeEpochCalculation*().

Recommended mitigation

Fetch the pending position change request directly from GMX when calculating the pool value. Such code exists in other option-trading projects.

Team response

Resolved

Mitigation review

Fixed by adding the amount in transit to the NAV if necessary.

TRST-M-2 BeyondPricer and GmxHedgingReactor are implicitly coupled to USDC collateral

- **Category:** Hard-coding issues
- **Source:** BeyondPricer.sol, GmxHedgingReactor.sol
- **Status:** Fixed

Description

BeyondPricer::quoteOptionPrice() returns the fee and premium in USDC decimals.

```
totalPremium = (premium.mul(_amount) + spread) / 1e12;
totalDelta = delta.mul(int256(_amount));
totalFees = feePerContract.mul(_amount);
```

In *GMXHedgingReactor::update*(), GMX's 30 decimal values are divided by 1e24.

Both of these are USDC dependent. If the **collateralAsset** changes to another token (e.g. DAI) these calculations are off by orders of magnitude.

Recommended mitigation

Calculate the divisors dynamically when setting the **collateralAsset**. Consider uncovering all instances of USDC-coupling by reviewing the code.

Team response

Resolved

Mitigation review

In BeyondPricer, decimals are dynamically checked from the ERC20 contract. In GmxHedgingReactor, decimal count is explicit as the constants **GMX_TO_COLLATERAL_DECIMALS** and **COLLATERAL_ASSET_DECIMALS** have been introduced.

TRST-M-3 Theoretical reentrancy in OptionRegistry::open() can lock funds

- **Category:** reentrancy flaws
- **Source:** OptionRegistry.sol
- **Status:** Fixed

Description

If **vaultId_** is zero for the provided option series a new vault is created. The new **vaultId_** is only set in the **vaultIds** mapping after the short tokens are minted and sent to msg.sender. If the function was to be reentered with the same series, yet another vault (with a higher vault ID) would be created for the same series and the **vaultIds** mapping would be overridden with the lower **vaultId_** before returning. The registry would permanently lose access to the higher vault ID.

```
if (vaultId_ == 0) {
    vaultId_ = (controller.getAccountVaultCounter(address(this))) + 1;
    vaultCount++;
}
uint256 mintAmount = OpynInteractions.createShort(
    gammaController,
    marginPool,
    _series,
    collateralAmount,
    vaultId_,
    amount,
    1
);
emit OptionsContractOpened(_series, vaultId_, mintAmount);
// transfer the option to the liquidity pool
SafeTransferLib.safeTransfer(ERC20(_series), msg.sender, mintAmount);
vaultIds[_series] = vaultId_;
```

Recommended mitigation

Follow the checks-effects-interaction pattern and move line 271 (below) to just after the checks on lines 255-258

```
vaultIds[_series] = vaultId_;
```

Team response

Resolved

Mitigation review

Recommended fix has been applied.

TRST-M-4 OptionExchange::_buyOption()/_sellOption() will not lead to update of LiquidityPool's ephemeralDelta

- **Category:** Logical Flaw
- **Source:** OptionExchange.sol
- **Status:** Fixed

Description

The liquidity pool's **ephemeralDelta** keeps track of the current exposure when the portfolio has not yet been updated via the periodic *fulfill()* call. There are two situations when the delta is leaked. In *_buyOption()*, any delta fulfilled using the existing OptionExchange exposure does not update it (because it does not go through *handlerWriteOption())*. In *_sellOption()*, any delta not delivered through the buyback mechanism doesn't update the ephemeral value. As a result, hedging may not be effective for prolonged periods.

Recommended mitigation

Expose LiquidityPool's *_adjustVariables()* and call it from the exchange in the described flows.

Team response

Resolved.

Mitigation review

Fixed with the suggested mitigation.

TRST-M-5 OptionExchange::_buyOption() may fail due to incorrect option balance assumption

- **Category:** Logical Flaw
- **Source:** OptionExchange.sol
- **Status:** Fixed

Description

In *buyOption()*, if there is existing long exposure registered in the portfolio, the exchange will directly sell oTokens to the sender instead of buying additional options through Opyn.

```
int256 longExposure =
portfolioValuesFeed.storesForAddress(buyParams.seriesAddress).longExp
osure;
uint256 amount = _args.amount;
emit OptionsBought(buyParams.seriesAddress, recipient, amount,
buyParams.premium, buyParams.fee);
if (longExposure > 0) {
   // calculate the maximum amount that should be bought by the user
   uint256 boughtAmount = uint256(longExposure) > amount ? amount :
uint256(longExposure);
```

```
    // transfer the otokens to the user
    SafeTransferLib.safeTransfer(
        ERC20(buyParams.seriesAddress),
        recipient,
        boughtAmount / (10**CONVERSION_DECIMALS)
    );
```

However, the exchange makes the assumption that whatever exposure is recorded in the portfolio is available as oTokens in the Exchange contract. That may not be the case when additional handlers store their own exposures in the portfolio. The impact is denial of service when buying options with **longExposure** > 0, and oToken balance is depleted in the exchange.

Recommended mitigation

Calculate the minimum transferrable tokens using the oToken balance of the exchange.

Team response

Resolved.

Mitigation review

Fixed by inserting a balance check.

TRST-M-6 OptionExchange::redeem() results in losses for liquidity providers from fees/slippage when converting non-USDC collateral

- **Category:** Logical Flaw
- **Source:** OptionExchange.sol
- **Status:** Acknowledged

Description

When an oToken's collateral is not USDC, *redeem()* will use a UniswapV3 compatible router to swap to USDC. However, this results in losses from fees/slippage for the liquidity provider which option buyers/sellers on the exchange potentially benefit from.

Recommended mitigation

Add a per-collateral premium / fee component for non-USDC collateralized assets, to reduce conversion related risks to the platform.

Team response

Acknowledged.

TRST-M-7 When changing position direction in GmxHedgingReactor::_changePosition() many calculations could be incorrect

- **Category:** Logical Flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

Several parts of the GMX reactor code assume that **internalDelta** is all delta in the current position i.e. there is one active position. However, the hedging reactor could get into a state where there were two open GMX positions, one long and one short.

This can happen when a decrease position request fails, but an increase position request succeeds. Since GMX position changes are not necessarily atomic, a GMX keeper could try to execute one position request in a different block to the other and a significant sharp price change could cause it to fail.

As a result of the two open GMX positions, functions such as *_adjustedReducePositionSize()* and *_getPositionSizeDeltaUsd()* would return incorrect results. This could lead to leaving a position in bad health, risking loss of funds.

Recommended mitigation

Consider simplifying the amount of possible states the contract may be in, by increasing a new position only after the previous position has been completely nullified.

Team response

Resolved.

Mitigation review

The fix introduces **openLongDelta**, **openShortDelta**, and **longAndShortOpen** variables to handle the multi-position state. When a new position is executed by GMX and the opposite direction position exists, **longAndShortOpen** is set to **true**. When a position drops to zero, it is set to **false.**

An issue still persists in the *update()* mechanism. New logic attempts to consolidate two opposite-directed positions.

```
if (_internalDelta >= 0) {
   // we are net long/neutral. close shorts
   uint256[] memory shortPosition = _getPosition(false);
   uint256 shortDelta = (shortPosition[0]).div(shortPosition[2]);
   collateralToRemoveShort = _getCollateralSizeDeltaUsd(false, false,
shortDelta, false);
   (bytes32 key1, int deltaChange1) = _decreasePosition(
      shortDelta,
      collateralToRemoveShort,
      false
   );
   decreaseOrderDeltaChange[key1] += deltaChange1;
   // then reduce longs by same delta
   collateralToRemoveLong = _getCollateralSizeDeltaUsd(false, false,
shortDelta, true);
   (bytes32 key2, int deltaChange2) = _decreasePosition(shortDelta,
collateralToRemoveLong, true);
   decreaseOrderDeltaChange[key2] += deltaChange2;
```

Essentially, it removes position and collateral from both sides, nullifying the short position in case the long one is greater. However, the highlighted line does not guarantee that the short position will be deleted. When calculating **shortDelta**, rounding will make the delta slightly

less than the effective delta. Later when decreasing by this delta, it will convert it to a position size which is slightly less than the real position size.

```
function _getPositionSizeDeltaUsd(
    uint256 _size,
    uint256 positionSize,
    bool _isLong
) private view returns (uint256) {
    return _size.mul(positionSize).div(_isLong ? openLongDelta :
openShortDelta);
}
```

As a result, *update()* may be perpetually stuck in the consolidation phase and not handle **isBelowMin** or **isAboveMax** scenarios, greatly increasing the risk of unhealthy positions.

Recommended mitigation

Use **openShortDelta** and **openLongDelta** instead of calculating them dynamically, when consolidating positions.

Mitigation review #2

The suggested fix has been applied.

TRST-M-8 Anyone can reset the GMX reactor's callback variables

- **Category:** Logical Flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

The variables **pendingIncreaseCallback** / **pendingDecreaseCallback** signal there is an uncompleted GMX request. The reactor supports a way to force its execution, with *executeIncreasePosition()* or *executeDecreasePosition()*.

In the mitigation review commit, code has changed and now the GMX call is wrapped in a try/catch.

```
function executeIncreasePosition(bytes32 positionKey) external {
    pendingIncreaseCallback = false;
    try gmxPositionRouter.executeIncreasePosition(positionKey,
payable(address(this))) {} catch {}
}
```

The issue is that when *executeIncreasePosition()* fails, for example if the function was called too quickly, the callback variable is still reset to **false**. Effectively, this allows anyone to disable the safety checks around these variables in a variety of GMX functions.

Recommended mitigation

Either set the function to be callable only by the keeper, or **do not** set the callback variables to **false** if an exception occurred.

Team response

Resolved

Mitigation review

The *execute()* functions no longer change state. The relevant state changes are only applied through the callback. Additionally, a governor-controlled recovery function has been added in case of state desynchronization.

TRST-M-9 When changing positions, GMX reactor can wrongly assume there's no pending callbacks

- **Category:** Logical Flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

The variables **pendingIncreaseCallback** / **pendingDecreaseCallback** signal there is an uncompleted GMX request. If there is a pending callback, *update*() and *hedgeDelta*() revert. The way in which two positions are handled in *update()* makes it possible that the callback is set to **false**, although the contract expects another callback.

update() calls *_decreasePosition()* twice, on opposite positions. When the first one completes, GMX calls *gmxPositionCallback()* on the reactor contract, which will set **pendingDecreaseCallback** to false. As there is still a pending position, it shouldn't be possible to call *hedgeDelta()* or *update()* at this moment, however that is not the case. Therefore it is possible that delta hedging would be incorrect, or that the positions would not be healthy.

```
if (isIncrease) {
    pendingIncreaseCallback = false;
    delete increaseOrderDeltaChange[positionKey];
    delete pendingIncreaseCollateralValue;
} else {
    pendingDecreaseCallback = false;
    delete decreaseOrderDeltaChange[positionKey];
}
```

Recommended mitigation

Only set the callback variable to false once **both** position requests have been executed.

Team response

Resolved

Mitigation review

Fixed by changing the callback variables to be **uint8** data type. Multiple decrease requests are handled correctly.

Low severity findings

TRST-L-1 GmxHedgingReactor::getPoolDenominatedValue() does not include pending decrease position under some conditions

- **Category:** Ordering assumptions
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

GMX has a two-step process for increasing/decreasing positions where a request is submitted and remains in a pending state until it is executed by a keeper. Sometimes delta will change so much that a call to **_changePosition** will submit both a decrease position and an increase position request to GMX to close out a short/long position and open a long/short position. However, the position increase could be executed first leaving the decrease still pending. The **internalDelta** could switch to the opposite sign affecting results returned by *_getPosition(internalDelta > 0).* The impact is that the original short/long position is not counted by *getPoolDenominatedValue()* leading to incorrect NAV evaluation in *executeEpochCalculation()*.

Recommended mitigation

Consider adding the value from both possible positions, rather than assuming only one is non-zero.

Team response

Resolved

Mitigation review

Recommended fix has been applied.

TRST-L-2 Unexpected leak of value when executing Opyn WithdrawCollateral or Settle actions

- **Category:** Logical Flaw
- **Source:** OptionExchange.sol
- **Status:** Fixed

Description

OptionExchange supports running multiple operations in succession, with funds storable in the exchange itself for the next action. In the post-execution check, any leftovers will be transferred back to the sender. Function *_runOpynActions()* performs a number of checks and effects before calling Opyn's *Controller::operate()* function with the relevant action. However, it doesn't check actions *OpenVault*, *WithdrawCollateral* or *Settle*. The latter two actions are similar enough to *WithdrawLongOption* and *MintShortOption* that they should

follow the same convention of checking whether **action.secondAddress == address(this)** and calling *_updateTempHoldings()* if true.

The impact is that a user could accidentally send tokens to the exchange thinking they would be handled with the same convention.

Recommended mitigation

Add else-if statements to handle the case where **action.secondAddress == address(this)**.

Team response

Resolved.

Mitigation review

Fixed by ensuring the destination of **WithdrawCollateral** and **SettleVault** actions is msg.sender.

TRST-L-3 Hedging reactors with delayed position updates (e.g. GmxHedgingReactor) will not be removed correctly by LiquidityPool::removeHedgingReactorAddress()

- **Category:** Logical Flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

Function *removeHedgingReactorAddress()* has the following lines:

```
if (delta != 0) {
   reactor.hedgeDelta(delta);
}
reactor.withdraw(type(uint256).max);
```

However, since GMX withdrawal is two step, the withdrawal needs to be delayed until the corresponding *decreasePosition*() call has been executed, at which point the funds are transferred to the reactor. Since this operation removes the reactor from the pool, there's a risk that the funds will be forever left in the reactor.

This issue has been classified as low as, if noticed, an easy fix would be to add and remove the reactor again once the pending position decrease goes through.

Recommended mitigation

Only allow removal of a reactor if it is not awaiting additional actions. Consider modifying *withdraw()* to revert when given **type(uint256).max** as an argument but there are still pending position changes. This should ensure backwards compatibility with the hedging reactor interface while allowing for the pending checks to be done.

Team response

Resolved.

Mitigation review

Withdrawals now validate that there is no pending increase or decrease position.

TRST-L-4 Calling GmxHedgingReactor::setPositionRouter() while there are pending GMX position changes will freeze functions

- **Category:** Logical Flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

If function *setPositionRouter()* gets called while there are pending create/decrease positions then **pendingIncreaseCallback** or **pendingDecreaseCallback** will remain true breaking functions *hedgeDelta()* and *update()*.

When GMX's PositionRouter calls function **_callRequestCallback** it will not revert even though the call to *gmxPositionCallback()* will revert (due to the access control check on line 790). This is because GMX uses a try/catch block in their code.

The impact is that GmxHedgingReactor can no longer be used to change positions leading to a lack of hedging functionality and capital being locked in GMX.

As it is unlikely that this bug would ever be triggered its risk has been assessed as low. Even if it did happen *setPositionRouter()* could be called on a fake contract that Rysk controls which could perform fake callbacks to reset the **pendingIncreaseCallback/pendingDecreaseCallback** variables. This would have to be done with caution to ensure that the changes to **internalDelta** caused by the call the *gmxPositionCallback()* mirrored what would have occurred as a result of the genuine callbacks from GMX.

Recommended mitigation

1. Don't allow *setPositionRouter()* to be called when **pendingIncreaseCallback/pendingDecreaseCallback** are not both false.
2. Allow for the callback values to be manually set back to false by admins.

Team response

Resolved.

Mitigation review

setPositionRouter() now validates that there is no pending increase or decrease position.

TRST-L-5 GmxHedgingReactor::update() can calculate incorrect values when there are pending position changes

- **Category:** Ordering assumptions
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

If *update()* is called again while there is still a pending increase/decrease, its call to *checkVaultHealth()* will be using the old value of **internalDelta**. However, the health *may* have changed which leads to a call to *_addCollateral()* when the previous call was to *_removeCollateral()* or vice versa. This may succeed in creating a new GMX position change request in the opposite direction even though *checkVaultHealth()* would have returned something different if the first call had already been processed.

Functions *_increasePosition() /_decreasePosition()* are protected from being called twice via the use of the **pending[Increase/Decrease]Callback** variables but nothing prevents two being open at the same time.

The impact is that the incorrect amount of collateral is added/removed. This has been assessed as low risk since it is unlikely that *update*() will be called this frequently, nor that the health of the position would change so quickly.

Recommended mitigation

Ideally, simplify state so that only one request can be active at a given time, regardless of direction.

Team response

Resolved

Mitigation review

State management has improved. The *update()* function will perform specialized logic when two positions are open at the same time. Also, *update()* cannot be called while there are pending position requests.

TRST-L-6 UniswapV3RangeOrderReactor does not comply with hedgeDelta() API

- **Category:** API issues
- **Source:** UniswapV3RangeOrderReactor.sol
- **Status:** Acknowledged

Description

The expected behavior of *hedgeDelta*() API is to increase the underlying exposure by the delta amount. However, UniswapV3RangeOrderReactor instead removes all previous exposure and hedges the input delta.

Recommended mitigation

Line up the *hedgeDelta*() behavior to be the same as all other reactors.

Team response

Acknowledged

TRST-L-7 Option pricing does not take into account all collateral allocated

- **Category:** Logical Flaw
- **Source:** BeyondPricer.sol
- **Status:** Acknowledged

Description

Option pricing takes into account the collateral allocation costs required to collateralize the oTokens minted by the registry. It is included in the spread value costs. However, the amount multiplied by the lending rate is the exact amount required to stay non-liquidateable. In fact, the collateral allocated by the pool is much higher, and comprised of two additional factors. The **bufferPercentage** variable defines the ratio between collateral allocated and collateral available in the pool, which cannot be crossed. In practice this makes the remaining portion of the collateral unusable. Additionally, the health factor should never actually be on the border of liquidation, as keepers ensure it is above a certain threshold (110-130%). These aspects make option pricing unattractive for LPs as most of the collateral is sitting idle and the premium does not reimburse them for it.

Recommended mitigation

When pricing options, consider calculating allocated collateral in a way that is accurate to the actual system functionality.

Team response

Acknowledged

TRST-L-8 Lack of safety checks when issuing existing options via issueNewSeries()

- **Category:** Logical Flaw
- **Source:** OptionCatalogue.sol
- **Status:** Acknowledged

Description

The function *issueNewSeries()* performs a check for whether an option series has already been issued.

```
// if the option is already issued then skip it
if (optionStores[optionHash].approvedOption) {
   continue;
}
```

However, it does not check whether the **isBuyable** and **isSellable** fields are the same or not. In the latter case the option series fields should be updated to the new ones.

Recommended mitigation

Add a check on **isBuyable**/**isSellable** and update them if they are different from before, or revert to be 100% sure the change is intentional.

Team response

Acknowledged

TRST-L-9 collateralAsset == strikeAsset assumption is not guaranteed

- **Category:** Logical Flaw
- **Source:** LiquidityPool.sol
- **Status:** Acknowledged

Description

It is a stated assumption that the incoming series' **collateralAsset == strikeAsset** in LiquidityPool. In *_issue*(), it checks that incoming asset is the same as the pool's asset.

```
// make sure option is being issued with correct assets
if (optionSeries.collateral != collateralAsset) {
   revert CustomErrors.CollateralAssetInvalid();
}
if (optionSeries.underlying != underlyingAsset) {
   revert CustomErrors.UnderlyingAssetInvalid();
}
if (optionSeries.strikeAsset != strikeAsset) {
   revert CustomErrors.StrikeAssetInvalid();
}
```

The assumption holds for the deployed pool as the two assets are the same, however other pool deployments could have different assets. This would break several assumptions in LiquidityPool calculations.

Recommended mitigation

Add a check that incoming **collateralAsset == strikeAsset**, or refactor code that assumes it holds.

Team response

Acknowledged

TRST-L-10 GmxHedgingReactor::sweepFunds only sweeps ETH

- **Category:** Logical Flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Acknowledged

Description

The *sweepFunds()* function allows the governor to pull any ETH in the contract. However, since the reactor also may hold other tokens such as collateral, it is advisable to support rescuing of ERC20 tokens as well.

Recommended mitigation

Support sweeping of collateral tokens by the governor.

Team response

Acknowledged

TRST-L-11 LiquidityPool::executeEpochCalculation may fail depending on withdraw behaviour

- **Category:** Logical Flaw
- **Source:** LiquidityPool.sol
- **Status:** Acknowledged

Description

The following code in LiquidityPool suggests that withdrawing amount *x* may result is withdrawing more than *x*.

```
amountNeeded -=
IHedgingReactor(hedgingReactors_[i]).withdraw(amountNeeded);
if (amountNeeded <= 0) {
   break;
}
```

However, **amountNeeded** is defined as a *uint256* variable so if withdraw allows this behavior the code would revert.

Recommended mitigation

If *withdraw()* aims to support over-withdrawals, change the **amountNeeded** data type to *int256.*

Team response

Acknowledged

TRST-L-12 GmxHedgingReactor's internalDelta is trusted but could be stale

- **Category:** Logical Flaw
- **Source:** GmxHedgingReactor.sol
- **Status:** Fixed

Description

The **internalDelta** variable is used for important calculations such as getting the position size and collateral size change, as well as determining if the current position is short or long. However, if liquidation occurs the true delta can change without this variable being updated.

Recommended mitigation

It's important that any code that uses **internalDelta** calls *sync()* first, to make sure it uses correct data.

Team response

Resolved.

Mitigation review

Recommended mitigation has been applied.

TRST-L-13 Waiving fees in OptionExchange leads to misaligned incentives

- **Category:** Incentives issues
- **Source:** OptionExchange.sol
- **Status:** Fixed

Description

Fees are waived if they are over 12.5% of the premium.

```
if ((sellParams.premium >> 3) > sellParams.fee) {
   SafeTransferLib.safeTransfer(ERC20(collateralAsset), feeRecipient,
sellParams.fee);
} else {
   // if the total fee is greater than premium / 8 then the fee is
waived, this is to avoid disincentivising selling back to the pool
for collateral release
   sellParams.fee = 0;
}
```

However, this has two negative side effects. Obviously less fees are generated but more importantly it incentivizes holding oTokens until the premium is low enough to waive the fees. It creates a step-function discontinuity whereby if the premium drops a small amount the overall amount the user receives will be higher. This incentivizes waiting for the premium to drop.

Recommended mitigation

It would probably be best to cap fees at the maximum amount, i.e. 12.5%, rather than waive the fee.

Team response

Resolved

Mitigation review

Suggested fix has been applied.

Additional recommendations

Add validation to constructor parameters

There is a general lack of validation on parameters to contract constructors. The most common omission is not checking that provided address are not equal to zero.

Add checks for all possible values of enumerations

The Rysk codebase makes frequent use of enumeration types. Consider the example below:

```
} else if (actionType == IController.ActionType.Liquidate) {
    revert ForbiddenAction();
} else if (actionType == IController.ActionType.Call) {
    revert ForbiddenAction();
}
_opynArgs[i] = action;
```

If the ActionType enum will ever be extended for additional types by Opyn, they will be transparently passed to the Opyn controller without filtering. It is considered bad practice to have a permissive catch-all case when handling enums.

Multiple copies of formatStrikePrice() function

Consider moving functions that are implemented in the same way across different contracts to a common utility library.

Documentation errors

In *OptionRegistry::getCollateral()*:

```
/**
 * @notice Send collateral funds for an option to be minted
 * @dev series.strike should be scaled by 1e8.
 * @param  series details of the option series
 * @param  amount amount of options to mint always in e18
 * @return amount transferred
 */
function getCollateral(Types.OptionSeries memory series, uint256
amount)
    external
    view
    returns (uint256)
{
```

The function is a view and doesn't send funds. It only calculates the amount.

In *LiquidityPool::deposit():*

```
/**
 * @notice function for adding liquidity to the options liquidity
pool
 * @param _amount    amount of the strike asset to deposit
 * @return success
 * @dev    entry point to provide liquidity to dynamic hedging vault
 */
function deposit(uint256 _amount) external whenNotPaused nonReentrant
returns (bool) {
```

Documentation states it is transferring **strikeAsset**, however it actually transfers **collateralAsset**.

```
SafeTransferLib.safeTransferFrom(collateralAsset, msg.sender,
address(this), _amount);
```

In *GmxHedgingReactor::changePosition():*

```
// remove the adjustedPositionSize from _amount to get remaining
amount of delta to hedge to open shorts with
_amount = _amount - int256(adjustedPositionSize);
```

Calculated amount is actually the amount to *open longs with*.

Use of magic numbers

There are many occurrences of magic numbers in the code base e.g. 1e24, 11000, 1e18, 1e12 in GmxHedgingReactor. There is no run-time cost to declaring them as constants and it can only improve readability and resistance to errors when making changes.

Naming of functions

LiquidityPool::getBalance() deducts **partitionedFunds**, but it's only correct to deduct if the parameter **asset** is **collateralAsset**. Consider renaming to *getCollateralBalance().*

Lack of event emission

Some functions such as *setPricer()* and *setOptionCatalogue()* don't emit an event which harms the visibility of the protocol.

Parsing safety checks

In parsing of combined actions into Rysk / Opyn actions, consider enforcing that unused arguments are zeroes, to protect against user errors and further limit the attack surface.

Separate safety checks

In *OptionExchange::_checkHash()* many safety checks are done that have nothing to do with the hash. It is better to separate them into appropriately named functions to improve readability and modifiability of the contract.

Accounting precautions

In *_sellOption()*, **tempHoldings** is set to the whole **heldTokens** and then **heldTokens** is decremented by up to amount. It's best to store in **tempHoldings** only the decrement amount, because **tempHoldings** is deducted from in different places and it should never allow deducting more than **_args.amount**.

Removing unused code

Several functions and structures are unused. Consider removing them to cut down the code size and improve readability.

Unrecommended usage of PRBMath functions for non-PRB numbers

In *AlphaPortfolioValuesFeed::updateStores()* the following code is used:

```
if (uint256(netDhvExposure[oHash].abs()) > maxNetDhvExposure) revert
MaxNetDhvExposureExceeded();
```

abs() is used from PRBMathSD59x18, which is a fixed point operations library, but the exposure is an *int256*. Luckily, *abs*() is implemented safely for integer numbers so there is no impact. Consider abandoning similar uses of fixed-point libraries for integers.

Naming conventions for public functions

The *LiquidityPool::_getVolatilityFeed()* function is marked as public. It is bad practice to begin a public function name with an underscore.

Naming mismatch isBuy/isSell

BeyondPricer::quoteOptionPrice() receives the **isSell** parameter, which is for the case that the *user* is selling options. It is then passed to *quotePriceGreeks()*, which accepts it as **isBuy**. The function operates correctly, but the terminology mixes between the user and the vault's perspective of the trade direction.

Mitigating reentrancy risks

OptionRegistry::registerLiquidatedVault() will get the liquidation details, update the collateral on the liquidity pool and then clear the liquidation details. This makes it theoretically vulnerable to over-updating of the collateral if attacker was to re-enter before clearing. It is best to clear the vault immediately.

Passing incorrect slippage value in GmxHedgingReactor::_increasePosition()

The **minOut** parameter of *increasePosition()* is always set to min WETH swap output, however when increasing short positions no swap is done from collateral, so it should set that parameter to zero.

Use of variables instead of literals

In *_changePosition(),* if the GMX reactor first decreases the original position, it passes **closedOppositeSideFirst** instead of passing false. Logically it is best to pass constants or literals when the value should not ever be different.

GMX hedgeDelta() does not fulfill API

hedgeDelta() should return only the delta change immediately active, as is done in UniswapV3RangeOrderReactor:

```
// satisfy interface, delta only changes when range order is filled
return 0;
```

However, the GMX reactor activity is not immediate (it can still fail if the second step fails), but the planned delta is already returned.

rebalancePortfolioDelta() ignores actual delta executed

The *hedgeDelta*() API returns the actual delta executed. When rebalance is called, the return value is ignored, possibly causing confusion around the real delta. Additionally, if called from Manager.sol, the intended delta counts for the keeper's limit, rather than the one executed.

```
deltaLimit[msg.sender] -= absoluteDelta;
liquidityPool.rebalancePortfolioDelta(delta, reactorIndex);
```

migrateOTokens() can migrate all types of tokens

In OptionExchange, *migrateOTokens*() is used to move ERC20 oTokens to the next exchange. There is a slight overprivilege concern as this function also allows transfer of any ERC20 token, including the collateral token which should only be vacated for the liquidity pool. Consider verifying that the token is an oToken.

Centralization risks

Governance pricing strategy is trusted

The SABR and other parameters, which eventually determine option pricing, are dynamically controlled by governance. LPs should be aware that an inadequate strategy would result in losses for the protocol.

GMX solvency risks

The project sends funds to external projects such as GMX in order to hedge its position. It should be noted that bugs, attacks or plain sharp price movements may tip the exchange into insolvency which would erase user funds.

Compromised owner risks

Despite the protocol not being upgradeable, there are still ways a compromised multisig can withdraw the entirety of the protocol funds, by changing trusted addresses such as the OptionRegistry's **liquidityPool**, and Protocol's **accounting**, to malicious addresses. The OptionRegistry can also be drained using the *migrateOtokens()* function.