Rysk

Smart Contract Security Assessment

December 9, 2022

ABSTRACT

Dedaub was commissioned to perform an audit of the new GMX Hedging Reactor of the Rysk protocol.

The audit report covers commit hash 96bc9aa5ba596196567438ee7cd20b6b6704952c. Two auditors worked on the codebase over 4 days. The scope of the audit is limited to a single file, GmxHedgingReactor.sol. Hedging reactors are contracts used by the Liquidity pool of Rysk to neutralize its total delta using derivatives. GMX hedging reactor opens and closes short and long positions on GMX and checks that these positions are supported by sufficient collateral.

One high severity and several medium and low severity issues were identified. All the issues have been taken care of by the protocol team.

SETTING & CAVEATS

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification.

Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. The scope of the audit includes smart contract code. Interactions with off-chain (front-end or back-end) code are not examined other than to consider entry points for the contracts, i.e., calls into a smart contract that may disrupt the contract's functioning.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third-party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming error.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	gmxPositionCallback is callable by anyone	**RESOLVED**

The function `GmxHedgingReactor::gmxPositionCallback`, which updates the `internalDelta` storage variable of the reactor as part of the two step process used by GMX to execute positions, does not impose a limit on the `msg.sender` or the amount of times that it may be called. An attacker could exploit this to completely mess up the reactor’s `internalDelta`.

A requirement that the `msg.sender` is the `gmxPositionRouter` should be added, as this would ensure that the `gmxPositionCallback` call comes after a successful call to `gmxPositionRouter::execute(Increase/Decrease)Position` made by a GMX keeper.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	GmxPositionRouter has unlimited spending approval	**RESOLVED**

In the `GmxHedgingReactor` constructor the `gmxPositionRouter` is approved to spend an infinite amount of `_collateralAsset`. It appears that this is unneeded and potentially dangerous, as the transfer of `_collateralAsset` is actually handled by the

GMX router, which gets approved for the exact amount needed in the function _increasePosition, and not by the gmxPositionRouter.

M2	Inconsistent returns in _changePosition	**RESOLVED**

The function GmxHedgingReactor::_changePosition is not consistent with the values it returns. Even though it should always return the resulting difference in delta exposure, it does not do so at the end of the if-branch of the if (_amount > 0) { … } statement. If the control flow reaches that point, it jumps at the end of the function leading to 0 being returned, i.e., as if there was no change in delta.

```
function _changePosition(int256 _amount) internal returns (int256) {
  // ..
  if (_amount > 0) {
    // ..
    // Dedaub: last statement is not a return
    increaseOrderDeltaChange[positionKey] += deltaChange;
  } else {
    // ..
    return deltaChange + closedPositionDeltaChange;
  }
  return 0;
}
```

We would suggest the following fixes:

```
function _changePosition(int256 _amount) internal returns (int256) {
  // ..
  if (_amount > 0) {
    // ..
    return deltaChange + closedPositionDeltaChange;
  } else if (_amount < 0) {
    // ..
    return deltaChange + closedPositionDeltaChange;
  }
```

```
    return 0;
}
```

Currently the return value of `_changePosition` is further returned by the function `hedgeDelta` and remains unused by its callers. However, this could change in future versions of the protocol leading to bugs.

M3 GMX long and short positions could co-exist

RESOLVED

GMX treats longs and shorts as completely separate positions, and charges borrowing fees on both simultaneously, thus the reactor deals with positions in such a way that ensures only a single position is open at a certain time. Nevertheless, due to the two-step process that GMX uses to create positions and the fact that the reactor does not take into account that a new position might be created while another one is waiting to be finalized, there exists a scenario in which the reactor could end up with a long and a short position at the same time. The scenario is the following:

1. Initially, there are no open positions
2. A long or short position is opened on GMX but is not executed immediately, i.e., `GmxHedgingReactor::gmxPositionCallback` is not called.

   The LiquidityPool reckons that a counter position should be opened and calls GmxHedgingReactor::hedgeDelta to do so.
3. When the two position orders are finally executed by GMX the reactor will have a long and a short position open simultaneously.

The above scenario might not be likely to happen as it requires the LiquidityPool to open two opposite positions in a very short period of time, i.e., before the first position order is executed by a GMX keeper or a keeper of the protocol. Nevertheless, we believe it would be better to also handle such a scenario, as it could mess up the reactor's accounting and the fix should be relatively easy.

<table>
<tr><td>M4</td><td>_getCollateralSizeDeltaUsd() in some cases underestimates the extra collateral needed for an increase of a position</td><td><b>ACKNOWLEDGED</b></td></tr>
<tr><td colspan="3">Whenever the hedging reactor asks for an increase of a position, <code>_getCollateralSizeDeltaUsd()</code> computes the extra collateral needed using <code>collateralToTransfer</code> (collateral needed to be added or removed from the position before its increase, to maintain the health to the desired value) and <code>extraPositionCollateral</code> (the extra collateral needed for the increase of the position).If <code>isAboveMax==true</code> and <code>extraPositionCollateral > collateralToTransfer</code>, then the collateral which is actually added is just <code>totalCollateralToAdd= extraPositionCollateral - collateralToTransfer</code>, which could be not sufficient to collateralize the increased position.
Let us try to explain this with an example. Suppose that initially there is a long position with <code>position[0]=10_000</code>, <code>position[1]=5_000</code>. Hedging reactor then asks for an increase of its position by 11_000. <code>extraPositionCollateral</code> will be 5_500. Suppose than in the meantime this position had substantial profits i.e. positive unrealised pnl=5_000. colateralToTransfer will be 5_000 and totalCollateralToAdd will be 5_500-5_000=500. Therefore the “leverage without pnl“ of the new position will be (10_000+11_000)/(5_000+500)=21_000/5_500=3.8. If this scenario is repeated, it could lead to the liquidation of the position.
We suggest adding a check that the total size of the position does not exceed its total collateral times maxLeverage, similar to the one used in the case of decreasing a position.</td></tr>
</table>

LOW SEVERITY:

ID	Description	STATUS
L1	setPositionRouter does not remove the old PositionRouter	**RESOLVED**

The function `GmxHedgingReactor::setPositionRouter` sets `gmxPositionRouter` to the new GMX PositionRouter contract that is provided and calls `approvePlugin` on the GMX Router contract to approve it. It does not revoke the approval to the old PositionRouter contract, which from now on is irrelevant to the reactor, by calling the function `denyPlugin` of the GMX Router contract.

L2	Potential underflow in CheckVaultHealth	**RESOLVED**

If a position is in loss, the formula of the health variable is the following one:

```
// GmxHedgingReactor.sol::_getCollateralSizeDeltaUsd():344
health=(uint256((int256(position[1])-int256(position[8])).div(int256(position[0]))) * MAX_BIPS) / 1e18;
```

There is no check if the difference (int256(position[1])-int256(position[8])) in the above formula is positive or not. It is possible, under specific economic conditions (and if the GMX Liquidators are not fast enough), that the result of this difference is negative. In such a case, the resulting value will be erroneous because of an underflow error.

Even if this scenario is not expected to happen on a regular basis, we suggest adding a check that this difference is indeed positive and if it is not extra measures should be taken to avoid liquidations.

Note that the same issue appears in getPoolDenominatedValue, leading to the execution reverting if an underflow occurs.

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	getPoolDenominatedValue() wastes gas	INFO

The function GmxHedgingReactor::getPoolDenominatedValue wastes gas by calling the function checkVaultHealth to retrieve just the currently open GMX position instead of directly calling the _getPosition function.

A2	gmxPositionCallback() can be made more gas efficient	INFO

The function GmxHedgingReactor::gmxPositionCallback is responsible for updating the internalDelta of the reactor with the values that are stored in the mappings increaseOrderDeltaChange and decreaseOrderDeltaChange. These mappings are essentially used as temporary storage before the change in delta is applied to the internalDelta storage variable. Thus, after a successful update the associated mapping element should be deleted to receive a gas refund for freeing up space on the blockchain.

A3	sync() can be made more gas efficient	INFO

The function GmxHedgingReactor::sync is implemented to consider the scenario where a long and a short position are open on GMX at the same time.

```
function sync() external returns (int256) {
  _isKeeper();
  uint256[] memory longPosition = _getPosition(true);
  uint256[] memory shortPosition = _getPosition(false);
  uint256 longDelta = longPosition[0] > 0 ?
```

```
    (longPosition[0]).div(longPosition[2]) : 0;
  uint256 shortDelta = shortPosition[0] > 0 ?
    (shortPosition[0]).div(shortPosition[2]) : 0;
  internalDelta = int256(longDelta) - int256(shortDelta);
  return internalDelta;
}
```

However, the reactor in whole is implemented in a way that ensures that a long and a short position cannot co-exist. Thus, the sync function can be implemented to take into account only the current open position, making it more efficient in terms of gas usage.

A4 Duplicate computations INFO

In `GmxHedgingReactor::_getCollateralSizeDeltaUsd` there is the following code:

```
// GmxHedgingReactor.sol::_getCollateralSizeDeltaUsd():670
if (
  int256(position[1] / 1e12) - int256(adjustedCollateralToRemove) <
  int256(((position[0] - _getPositionSizeDeltaUsd(_amount, position[0])) /
1e12) / (vault.maxLeverage() / 11000))
) {
  adjustedCollateralToRemove =
    position[1] / 1e12 -
    ((position[0]-_getPositionSizeDeltaUsd(_amount,position[0])) / 1e12) /
    (vault.maxLeverage() / 11000);
  if (adjustedCollateralToRemove == 0) {
    return 0;
  }
}
```

Observe that the quantity `(position[0]-_getPositionSizeDeltaUsd(_amount, position[0])) / 1e12) / (vault.maxLeverage() / 11000)` is computed twice which can be avoided by computing it once and storing its value to a local variable. The same is true for the quantity _amount.mul(position[2] / 1e12).div(position[0] / 1e12) that appears twice in the following computation:

```
// GmxHedgingReactor.sol::_getCollateralSizeDeltaUsd():651
collateralToRemove =
  (1e18 -
    (
      (int256(position[0]/1e12)+int256((leverageFactor.mul(position[8]))/1e12))
        .mul(1e18-int256(_amount.mul(position[2]/1e12).div(position[0]/1e12)))
        .div(int256(leverageFactor.mul(position[1])/1e12))
    )).mul(int256(position[1]/1e12)) -
  int256(_amount.mul(position[2]/1e12).div(position[0]/1e12)
    .mul(position[8]/1e12));
```

The above computation can be simplified even further by applying specific mathematical properties:

```
uint256 d = _amount.mul(position[2]).div(position[0]);
collateralToRemove =
  (int256(position[1] / 1e12) - (
    ((int256(position[0]) + int256(leverageFactor.mul(position[8]))) / 1e12)
      .mul(1e18 - int256(d)).div(int256(leverageFactor))
  )) - int256(d.mul(position[8] / 1e12));
```

A5 Duplicate calls INFO

The functions _increasePosition and _decreasePosition of the reactor unnecessarily call gmxPositionRouter's minExecutionFee function twice each instead of caching the returned value in a local variable after the first call.

A6 Incorrect comment in _increasePosition INFO

The comment describing the parameter _collateralSize of the function _increasePosition should read “amount of collateral to add” instead of “amount of collateral to remove”.

A7	Unused errors	INFO

The following errors are defined but not used:

```
// GmxHedgingReactor.sol::_getCollateralSizeDeltaUsd():88
error ValueFailure();
error IncorrectCollateral();
error IncorrectDeltaChange();
error InvalidTransactionNotEnoughMargin(int256 accountMarketValue, int256
totalRequiredMargin);
```

A8	Compiler bugs	INFO

The code is compiled with Solidity 0.8.9, which, at the time of writing, has some known bugs, which we do not believe to affect the correctness of the contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.