MiloTruck

Rodeo Finance

Security Review

December, 2023

Contents

Introduction

About MiloTruck

MiloTruck is an independent security researcher who specialises in smart contract audits. Currently, he works as a Senior Auditor at Trust Security and Associate Security Researcher at Spearbit. He is also one of the top wardens on Code4rena.

For private audits or security consulting, please reach out to him on:

- Twitter - @milotruck

You can also request a quote on Code4rena or Cantina to engage them as an intermediary.

Disclaimer

A smart contract security review **can never prove the complete absence of vulnerabilities**. Security reviews are a time, resource and expertise bound effort to find as many vulnerabilities as possible. However, they cannot guarantee the absolute security of the protocol in any way.

Risk Classification

Severity Level

	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality.
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality.
- Low - Funds are **not** at risk.

Likelihood

- High - Highly likely to occur.
- Medium - Might occur under specific conditions.
- Low - Unlikely to occur.

Executive Summary

About Rodeo Finance

Rodeo is a decentralised finance protocol that allows its community to earn yield on a range of managed and passive investment strategies on Arbitrum and the greater DeFi ecosystem. At its core, Rodeo offers Boosted Yield Farming and Structured Yield Products.

Boosted Yield Farming allows liquidity providers and farmers who seek boosted yield to stake their yield bearing assets for use in the Rodeo Farms, thus maintaining exposure to the underlying asset and earning additional yields in the farms.

Structured Yield Products allow Rodeo to innovate on top of existing DeFi infrastructure to create novel products to boost, leverage, automate and capture the best DeFi opportunities and narratives.

Overview

Project Name	Rodeo Finance (GMX GM Strategy)
Project Type	Yield, GMX V2 Integration
Repository	https://github.com/rodeofi/rodeo
Commit Hash	958715567457ef93b1f99048d28f33418a681502

Scope

211 SLOC

- contracts/src/strategies/StrategyGMXGM.sol

Issues Found

Severity	Count
High	4
Medium	4
Low	3
Informational	1

Findings

Summary

ID	Description	Severity
H-01	`tma == 0` check can be skipped by donating assets	High
H-02	Pulling assets in before calling `rate()` inflates `tma` in `_mint()`	High
H-03	Share calculation in `_mint()` is susceptible to inflation attacks	High
H-04	`MAX_PNL_FACTOR` is invalid for fetching GM token prices	High
M-01	Unsafe cast of GM token price in `marketTokenPrice()`	Medium
M-02	Missing functionality to handle execution fee refunds	Medium
M-03	Missing slippage checks for GMX withdrawals	Medium
M-04	Fetching decimals for index tokens will not work for some markets	Medium
L-01	GMX's `exchangeRouter` and `reader` addresses can change	Low
L-02	Avoid hardcoding `callbackGasLimit` in `_earn()`	Low
L-03	Strategy contract might miss out on token airdrops	Low
I-01	Code in `_rate()` can be simplified	Informational

High Severity Findings

H-01: `tma == 0` check can be skipped by donating assets

Description

In `_mint()`, when there are currently no assets in the strategy (i.e. `tma == 0`), the number of shares returned is equal to `val`:

StrategyGMXGM.sol#L86

```
return tma == 0 ? val : val * tot / tma;
```

However, `tma` is equal to `rate(tot)`, which represents the total amount of long, short and market tokens currently held by the contract:

StrategyGMXGM.sol#L189-L197

```
function _rate(uint256 sha) internal view override returns (uint256) {
    uint256 bal = IERC20(tokenLong).balanceOf(address(this));
    uint256 val = strategyHelper.value(tokenLong, bal);
    bal = IERC20(tokenShort).balanceOf(address(this));
    val += strategyHelper.value(tokenShort, bal);
    bal = IERC20(market).balanceOf(address(this)) + amountPendingDeposit + amountPendingWithdraw;
    val += (bal * marketTokenPrice() / 1e18);
    return sha * val / totalShares;
}
```

As such, by directly transferring 1 wei of any token into the contract, an attacker can make `tma` become non-zero and bypass the `tma == 0` check.

If this occurs while `tot` is still 0 (e.g. before the first deposit), all subsequent deposits will mint 0 shares as `val * tot / tma` will always be equal to 0, causing a loss of funds for depositors.

Recommendation

Check if the total amount of shares is non-zero as well:

```
- return tma == 0 ? val : val * tot / tma;
+ return (tma == 0 || tot == 0) ? val : val * tot / tma;
```

Rodeo Finance: Fixed as recommended in commit de3e3ad.

H-02: Pulling assets in before calling `rate()` inflates `tma` in `_mint()`

Description

`_mint()` transfers `ast` into the contract using `pull()` first, and calls `rate(tot)` to get the total value of long, short and market tokens held in the contract afterwards:

StrategyGMXGM.sol#L78-L81

```
pull(IERC20(ast), msg.sender, amt);
uint256 slp = getSlippage(dat);
uint256 tot = totalShares;
uint256 tma = rate(tot);
```

However, `rate()` uses the current balance of `tokenLong` and `tokenShort` to determine the total value of assets held in the contract.

As such, if `ast` happens to be the same token as either `tokenLong` or `tokenShort`, `rate(tot)` will incorrectly include the value of assets that was just transferred in. This will end up minting a smaller amount of shares to the depositor as `tma` is inflated in the following line:

StrategyGMXGM.sol#L86

```
return tma == 0 ? val : val * tot / tma;
```

For example:

- Assume `tokenShort` and `ast` are both USDC.e.
- The strategy currently contains 1000 `tokenShort` and 1000 shares.
- Bob deposits 1000 USDC.e. In `_mint()`:
  - `tot = 1000`, but `tma` is `2000` instead of `1000`
  - Bob receives only 500 shares, even though he should have received 1000.

Recommendation

Only transfer assets in after calling `rate(tot)`:

```
- pull(IERC20(ast), msg.sender, amt);
  uint256 slp = getSlippage(dat);
  uint256 tot = totalShares;
  uint256 tma = rate(tot);
+ pull(IERC20(ast), msg.sender, amt);
```

This ensures that `tma` will always exclude the value of tokens from the current deposit.

Rodeo Finance: Fixed as recommended in commit 3afb854.

H-03: Share calculation in `_mint()` is susceptible to inflation attacks

Description

`_mint()` calculates the number of shares to be minted to depositors as such:

StrategyGMXGM.sol#L80-L86

```
uint256 tot = totalShares;
uint256 tma = rate(tot);

IERC20(ast).approve(address(strategyHelper), amt);
uint256 bal = strategyHelper.swap(ast, tokenShort, amt, slp, address(this));
uint256 val = strategyHelper.value(tokenShort, bal);
return tma == 0 ? val : val * tot / tma;
```

Where:

- `tot` is the total number of shares.
- `tma` is the total value of assets currently held in the strategy.
- `val` is the value of assets deposited by the user.

However, this implementation is vulnerable to inflation attacks, where an attacker transfers assets directly to the strategy to increase `tma` while `tot` is small. For example:

- Assume the strategy currently has no assets and shares.
- Alice deposits an extremely small amount of tokens, such that `val` is `1` in `_mint()`:
  - This mints 1 share to Alice; `tot` is now `1`.
- Alice transfers 5000 USD worth of `tokenShort` directly to the strategy.
- Bob deposits 1000 USD worth of assets. In `_mint()`:
  - `tot = 1` and `val = 1000e18`
  - Since Alice "donated" 5000 USD worth of `tokenShort`, `tma` is now `5000e18`.
  - `val * tot / tma` rounds down to 0, causing Bob to receive 0 shares.
- All assets deposited by Bob accrues to Alice's 1 share.

In this scenario, any subsequent deposit that is smaller than `tma` will receive 0 shares. A first depositor can abuse this to steal funds from future depositors.

Recommendation

Consider setting `totalShares` to a small amount of shares in the constructor, when the strategy is first deployed.

This has the same effect as minting a small number of shares to a "dead" address on the first deposit, which is what Uniswap V2 does to prevent such inflation attacks.

Rodeo Finance: Fixed in commit `ff2e181` by initialising `totalShares` to `1_000_000` on deployment. Additionally, our team will always perform the first deposit before listing any strategy, so the loss from minting these `1e6` shares out of thin air will be incurred by us instead of users.

H-04: `MAX_PNL_FACTOR` is invalid for fetching GM token prices

Description

`marketTokenPrice()` calls the `getMarketTokenPrice()` function of GMX's reader with `pnlFactor` as `MAX_PNL_FACTOR`:

StrategyGMXGM.sol#L216-L219

```
bytes32 pnlFactor = keccak256(abi.encode("MAX_PNL_FACTOR"));
(int256 price,) = rdr.getMarketTokenPrice(
    dataStr, marketInfo, indexTokenPrice, longTokenPrice, shortTokenPrice, pnlFactor, false
);
```

However, this is not a valid key for `pnlFactor`. According to GMX's documentation, `pnlFactor` has to be one of the following three keys:

It is possible for the pending PnL to be capped, the factors used to calculate the market token price can differ depending on the activity:

- *Keys.MAX_PNL_FACTOR_FOR_DEPOSITS: this is the PnL factor cap when calculating the market token price for deposits*
- *Keys.MAX_PNL_FACTOR_FOR_WITHDRAWALS: this is the PnL factor cap when calculating the market token price for withdrawals*
- *Keys.MAX_PNL_FACTOR_FOR_TRADERS: this is the PnL factor cap when calculating the market token price for closing a position*

These different factors can be configured to help liquidity providers manage risk and to incentivise deposits when needed, e.g. capping of trader PnL helps cap the amount the market token price can be decreased by due to trader PnL, capping of PnL for deposits and withdrawals can lead to a lower market token price for deposits compared to withdrawals which can incentivise deposits when pending PnL is high.

As such, `marketTokenPrice()` will return an incorrect price for GM tokens. More specifically, the price returned would not factor in the PnL of the market, causing all accounting in the contract to be incorrect.

Recommendation

Modify `marketTokenPrice()` to use the keys listed above.

As seen from above, the recommended approach is to use `MAX_PNL_FACTOR_FOR_DEPOSITS` during deposits and `MAX_PNL_FACTOR_FOR_WITHDRAWALS` during withdrawals.

Rodeo Finance: Fixed as recommended in commit aa54aaa.

Medium Severity Findings

M-01: Unsafe cast of GM token price in `marketTokenPrice()`

Description

In `marketTokenPrice()`, the GM token price returned by `getMarketTokenPrice()` is cast to `uint256` directly:

StrategyGMXGM.sol#L217-L222

```
(int256 price,) = rdr.getMarketTokenPrice(
    dataStr, marketInfo, indexTokenPrice, longTokenPrice, shortTokenPrice, pnlFactor, false
);

// returned as 1e30, lets downscale to 1e18 used in here
return uint256(price) / 1e12;
```

According to GMX's documentation, although unlikely, it is possible for a market's token to have a negative price:

> *It is rare but possible for a pool's value to become negative, this can happen since the impactPoolAmount and pending PnL is subtracted from the worth of the tokens in the pool*

As such, casting directly to `uint256` is dangerous as `marketTokenPrice()` will end up returning an extremely large value when `price` is negative. Should this occur, it would inflate the value of GM tokens held by the strategy in `_rate()`, causing all calculations in the contract to become incorrect.

Recommendation

If `price` is negative, consider returning `0` as the price of GM tokens:

```
- return uint256(price) / 1e12;
+ return price < 0 ? 0 : uint256(price) / 1e12;
```

Rodeo Finance: Fixed as recommended in commit 74dbc96.

M-02: Missing functionality to handle execution fee refunds

Description

When creating deposits and withdrawals in GMX, the strategy sends an execution fee in ETH to GMX's `exchangeRouter` contract.

According to GMX's documentation, the excess execution fee will be refunded and sent back to the strategy contract.

> *This network cost is paid to the blockchain network when the order is executed. This cost is overestimated to handle potential increases in gas price, when the order is executed, the excess execution fee is sent back to your account.*

Note that this refund also occurs when deposits/withdrawals are cancelled.

As the strategy contract does not have a receive or payable fallback function, the transfer of ETH from GMX will revert. GMX will then wrap the ETH and send it as WETH instead, according to the integration notes:

> *ETH transfers are sent with NATIVE_TOKEN_TRANSFER_GAS_LIMIT for the gas limit, if the transfer fails due to insufficient gas or other errors, the ETH is send as WETH instead*

This behaviour can be observed in GasUtils.sol#L90-L136.

However, since the strategy contract has no functionality to handle execution fee refunds, the refunded WETH will be permanently stuck in the contract and cannot be withdrawn by anyone.

Recommendation

Consider handling execution fee refunds in the strategy contract. Some options are:

1. Add a receive function and a function to withdraw ETH from the contract.
2. Add a function to swap WETH to `tokenShort`. This will redistribute all execution fee refunds back to depositors.

Rodeo Finance: Fixed in commit `51fd9c9` by adding a `receive()` function and a new `withdrawEth()` function for admins to withdraw ETH from the strategy.

M-03: Missing slippage checks for GMX withdrawals

Description

When creating GMX withdrawals, `minLongTokenAmount` and `minShortTokenAmount`, which represent the minimum output amount of long and short tokens respectively, are set to `0`:

StrategyGMXGM.sol#L154-L166

```
IExchangeRouter.CreateWithdrawalParams memory params = IExchangeRouter.CreateWithdrawalParams({
    receiver: address(this),
    callbackContract: address(this),
    uiFeeReceiver: address(0),
    market: market,
    longTokenSwapPath: new address[](0),
    shortTokenSwapPath: new address[](0),
    minLongTokenAmount: 0,
    minShortTokenAmount: 0,
    shouldUnwrapNativeToken: false,
    executionFee: msg.value,
    callbackGasLimit: 500000
});
```

This leaves withdrawals vulnerable to slippage, especially since withdrawal requests are not executed instantly, but reliant on GMX's keepers to execute the request.

The market's state might change significantly between the time the withdrawal request was created and the time GMX's keepers execute the withdrawal, potentially causing the amount of tokens received to be much smaller than expected and resulting in a loss of funds.

Recommendation

Consider not leaving `minLongTokenAmount` and `minShortTokenAmount` as `0`.

A possible approach would be to fetch the output amount of long and short tokens with `Reader.getWithdrawalAmountOut()`, and use those values as the minimum amounts after factoring in slippage.

Rodeo Finance: Fixed as recommended in commit `da029f1`.

M-04: Fetching decimals for index tokens will not work for some markets

This finding was discovered by the Rodeo Finance team during the audit.

Description

`marketTokenPrice()` attempts to fetch the decimals for index tokens as such:

StrategyGMXGM.sol#L205-L206

```
uint256 price0 = hlp.price(marketInfo.indexToken) *
    (10 ** (30 - IERC20(marketInfo.indexToken).decimals())) / 1e18;
```

However, for some markets, the address stored at `marketInfo.indexToken` is not a contract. These markets are:

- Synthetic markets:
  - BTC / USD
  - DOGE / USD
  - LTC / USD
  - XRP / USD
- Stablecoin markets, where `indexToken == address(0)`:
  - USDC / USDC.e
  - USDC / USDT
  - USDC / DAI

For these markets, calling `decimals()` on the `indexToken` address will revert. As such, the strategy's current implementation is incompatible with these markets.

Recommendation

For synthetic markets, convert the `indexToken` price returned from Chainlink's price feeds to GMX's internal representation using the decimals given here. The price feeds used should match the ones used by GMX, with the exception of LTC / USD and XRP / USD.

For stablecoin markets, set `price0` to `0`.

Rodeo Finance: Fixed in commit c4ca7c8. `strategyHelper.price()` now supports synthetic markets (addresses without code) and added `indexTokenDecimals` to store the decimals of `indexToken` when it is not an ERC-20 contract. We do not intend to support stablecoin pools.

Low Severity Findings

L-01: GMX's `exchangeRouter` and `reader` addresses can change

Description

GMX's `exchangeRouter` and `reader` address is declared as immutable in the strategy contract:

StrategyGMXGM.sol#L15-L16

```
IExchangeRouter public immutable exchangeRouter;
IReader public immutable reader;
```

However, according to GMX's integration notes, these addresses will change if GMX decides to upgrade their functionality:

> *If using contracts such as the ExchangeRouter, Oracle or Reader do note that their addresses will change as new logic is added*

If either address changes, the strategy contract will no longer be functional and will have to be migrated to a new one.

Recommendation

Consider not declaring both addresses as immutable and adding permissioned functions to change them.

Rodeo Finance: Fixed as recommended in commit `770382f`. Note that even if these variables were unchangeable, we could upgrade the strategy contract and migrate assets over from the old strategy.

L-02: Avoid hardcoding `callbackGasLimit` in `_earn()`

Description

When creating deposit and withdrawal requests, `_earn()` sets `callbackGasLimit` to a hard-coded value of `500000`. However, this is dangerous as gas costs can potentially change, especially for `afterWithdrawalExecution()` where there is additional logic to swap `tokenLong` back to `tokenShort`.

Although unlikely, `callbackGasLimit` could possibly become insufficient to execute the callback functions in the future, which would make the strategy contract non-functional.

Recommendation

Consider setting `callbackGasLimit` as a state variable and adding a function to modify it.

Rodeo Finance: Fixed as recommended in commit `afe177b`.

L-03: Strategy contract might miss out on token airdrops

Description

According to GMX's integration notes, GMX might have airdrops to GM token holders:

> *Token airdrops may occur to the accounts of GM token holders, integrating contracts holding GM tokens must be able to claim these tokens otherwise the tokens would be locked, the exact implementation for this will vary depending on the integrating contract, one possibility is to allow claiming of tokens that are not market tokens, this can be checked using the* `Keys.MARKET_LIST` *value*

Should this ever occur, token airdrops to the strategy contract will be lost as it does not have functionality to transfer/swap tokens that are not `tokenShort` or `tokenLong`.

Recommendation

Consider adding functionality to handle token airdrops. One possible approach would be to add a function that swaps tokens to `tokenShort`, which would redistribute airdrops to depositors.

Rodeo Finance: Fixed in commit f4a5df7 by adding a new function for admins to withdraw any token except `market`, `tokenShort` or `tokenLong`.

Informational Findings

I-01: Code in `_rate()` can be simplified

`_rate()` can be re-written as such:

```
function _rate(uint256 sha) internal view override returns (uint256) {
    uint256 val = strategyHelper.value(tokenLong, IERC20(tokenLong).balanceOf(address(this)));
    val += strategyHelper.value(tokenShort, IERC20(tokenShort).balanceOf(address(this)));

    uint256 bal = IERC20(market).balanceOf(address(this)) + amountPendingDeposit + amountPendingWithdraw;
    val += (bal * marketTokenPrice() / 1e18);

    return sha * val / totalShares;
}
```

Rodeo Finance: Fixed as recommended in commit e6125d5.