Brahma Polygains Vault Audit Report

Summary

Auditor

Ivan Kuznetsov, https://twitter.com/jeiwan7

Project Type

DeFi, yield aggregator

Total Issues

8 (4 resolved)

High Severity Issues

0

Medium Severity Issues

5

Low Severity Issues

3

Scope

Repository

https://github.com/Brahma-fi/matic_vault

Commit hash

`c88fc4bafabe2f59fde284afae58ac16eb67fc79`

Mitigations Commit Hash

`4f20372c95c970f59d2e3ece0ef0d7174cae07f8`

Files in scope

- `./src/BalancerPositionHandler.sol`
- `./src/BalancerYieldExecutor.sol`
- `./src/BaseTradeExecutor.sol`
- `./src/CurvePositionHandler.sol`
- `./src/CurveYieldExecutor.sol`
- `./src/Harvester.sol`
- `./src/Vault.sol`

System Overview

The system that was audited is a DeFi yield aggregator on Polygon that allows users to deposit their `MATIC` and `WMATIC` tokens and earn passive income. To generate yield, the protocol integrates with Curve and Balancer pools, namely the `wMATIC/stMATIC` pools, and respective gauges. The protocol operates in epochs: yield is harvested and compounded once per epoch (although epoch length wasn't strictly defined at the moment of the audit).

`Vault` is the main contract via which users interact with the system. Funds can be deposited using the `deposit` function and withdrawn using the `withdraw`. Both deposits and withdrawals are subject to fees: deposit fees equal to the previous epoch's yield and protect against timed deposits aimed at benefiting from the distribution of yield; withdrawal fees (exit penalty) disincentivize withdrawal of liquidity and removal of liquidity from reward-generating gauges.

Deposited funds remain in the `Vault` contract until deposited into yield executors by the keeper. `BalancerYieldExecutor` and `CurveYieldExecutor` are the contracts that transfer funds between the `Vault` and Curve and Balancer vaults and gauges. Both contracts implement `depositIntoExecutor` and `withdrawfromExecutor` functions to deposit funds into Curve/Balancer vaults and gauges and withdraw them. `depositIntoExecutor` performs two operations:

1. transfers user funds into the `WMATIC/stMATIC` pool of Curve/Balancer and receives the LP tokens of the pools in return;
2. stakes the LP tokens in respective gauges.

`withdrawfromExecutor` performs the above two operations in the reverse order. However, before removing LP tokens from gauges, the function checks if the yield executor contract's balance is enough to cover the withdrawal. If so, the contract's LP tokens are transferred out. This is done to optimize the yield generated in the gauges.

The gauges the system integrates with generate yield in multiple tokens: `LDO`, in the Curve gauge; `BAL`, `LDO`, in the Balancer gauge. Generated yield is harvested and re-invested periodically by the keeper by calling the `Vault.harvest` function. Claiming yield from each of the yield executors involves two operations:

1. claiming rewards from respective gauge;
2. sending rewards to the `Harvester` contract;
3. calling the `Harvester.harvest` function.

The `Harvester.harvest` function seels the reward tokens for `WMATIC`. It integrates with two decentralized exchanges:

1. to sell `LDO`, it integrates with Uniswap V2;
2. to sell `BAL`, it integrates with Balancer.

After selling the reward tokens, `Harvester.harvest` sends proceedings back to the yield executor, which sends them to the vault. The `Vault.harvest` function then:

1. calculates the performance fee from the change of the share price (if the change is positive) and stores it in the contract (the governance can collect it at any time later on);

2. optionally, send the harvested yield to a trade executor, to boost the rewards (rewards boosting was out of scope of the audit).

Objectives

The objective of the audit was to:

1. identify all possible situations when users may lose their funds, lose access to them, get a wrong number of shares when depositing or don't receive shares at all, have their funds locked in the contracts or one of the contracts the system integrates with;
2. identify possible attack vectors on the system and find out if any of them can be executed;
3. identify and find workarounds for the known reentrancy attacks on Curve and Balancer pools.

Read-only Reentrancy Attacks on Curve Pools

At the moment of the audit, some Curve pools are known to be vulnerable to read-only reentrancy attacks (the attack is explained in details in this blog post by ChainSecurity).

The system implements the following mitigations:

1. the only Curve pool used by the system uses only ERC20 tokens (`WMATIC` and `stETH`), which don't trigger callbacks on transfers;
2. the only Curve pool used by the system doesn't use the native coin (`MATIC`), thus the pool doesn't call the caller when on withdrawals;
3. the system uses the `remove_liquidity_one_coin` function to withdraw funds, which makes only one external call (to transfer a token out) and doesn't cause a corrupted state of the pool.

Read-only Reentrancy Attacks on Balancer Pools

At the moment of the audit, some Balancer pools are known to be vulnerable to read-only reentrancy attacks, which exploit a bug in the `getRate` function of a pool. As per this Balancer recommendation, the system doesn't use ERC777 tokens or other tokens that execute a callback on transfers:

> WARNING: since this function reads balances directly from the Vault, it is potentially subject to manipulation via reentrancy. However, this can only happen if one of the tokens in the Pool contains some form of callback behavior in the `transferFrom` function (like ERC777 tokens do). These tokens are strictly incompatible with the Vault and Pool design, and are not safe to be used.

On-chain Slippage Tolerance Protection

The system uses on-chain slippage tolerance protection calculations. To avoid token price manipulations, the system relies on methods that are not vulnerable to manipulations:

1. for Curve pools, the system uses the `lp_price` function and the `price_oracle` function of the Curve pool to price LP tokens;
2. for Balancer pools, the system uses the `getRate` function and the `getScalingFactors` function of the Balancer pool to price LP tokens.

The `lp_price` and `getRate` functions return the invariant of a pool; the `price_oracle` function returns the internal price of a pool; `getScalingFactors` returns the prices as reported by a Balancer's rate provider (a price oracle).

Findings

[Medium-01] Anyone can withdraw funds from gauges, breaking the yield generation mechanism

Targets

- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Vault.sol#L160

Impact

Anyone may drain funds from yield executors, resulting in no yield generation for the depositors of the vault.

Proof of Concept

Funds deposited to the vault can be stored in multiple places:

1. in the vault itself;
2. on, in one or many yield executors. Thus, the withdrawing mechanism needs to allow users to withdraw their funds no matter whether they're stored in the vault or in one of the yield executors. When withdrawing funds:
3. an amount of funds proportional to the amount of shares withdrawn is transferred from the vault (Vault.sol#L153-L155);
4. amounts of funds proportional to the amount of shares withdrawn are transferred from the yield executors (Vault.sol#L158-L169).

This implementation assumes that the funds withdrawn were previously deposited to the yield executors. However this is not guaranteed, as depositing funds to yield executors is asynchronous and is executed at the discretion of the keeper. This allows the following exploit scenario:

1. Suppose that the vault has 1000e18 shares minted after 1000e18 funds were deposited. Also suppose that 500e18 of the deposited funds were transferred to an executor by the keeper. Thus, the current vault's balance is 500e18 and the yield executor's balance is also 500e18.
2. A new depositor deposits 1000e18 of funds to the vault, they receive 1000e18 shares. The current vault's balance is 1500e18 and the number of shares is 2000e18 (the amount deposited by the depositor equals to `totalVaultFunds`; `lastEpochYield` is zero). The keeper hasn't transferred the funds to the yield executor yet.
3. The depositor withdraws 1000e18 of shares from the vault. They will receive:
    1. from the vault: `100018 * 1500e18 / 2000e18 = 750e18`
    2. from the yield executor: `1000e18 * exec.totalLPTokens / 2000e18 = exec.totalLPTokens / 2`, or a half of the yield executor's balance, which is 250e18.

4. In total, the depositor will receive `750e18 + 250e18 = 1000e18`, i.e. their initial deposit amount. However, the balance of the yield executor will be reduced by 250e18, which decreases the rewards for other depositors of the vault.
5. The withdrawal may be repeated multiple times to bring the amount of funds in the yield executor to 1 wei.

Recommended Mitigation Steps

Before withdrawing from yield executors, consider computing the total amount of funds to be withdrawn (include both vault's balance and yield executors' balances; exclude pending fee) and checking if the entire amount may be withdrawn from the vault's balance alone: if the vault has enough funds, withdraw funds only from the vault.

[Medium-02] Pending yield is not excluded from share price calculation in `harvest`

Targets

- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Vault.sol#L215
- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Vault.sol#L228-L234

Impact

Yield may be counted twice in the share price calculation in the `harvest` function: first, as `baseYield`; then, after being transferred to the `tradeExecutor` and withdrawn in the next epoch, as `boostedYield`. Thus, the performance fee may be applied to pending yield (the yield to be boosted). Also, after `lastEpochYield - fee` has been transferred to `tradeExecutor`, the share price will be different than the one used to calculate the performance fee, due to reduced balance of the vault.

Proof of Concept

In the harvest function, the current share price is calculated as the ratio of the total vault funds and the total supply of shares. Total vault funds is the sum of:

1. the current vault's balance;
2. the funds stored on yield executors.

At the moment when the current share price is calculated, the current vault's balance includes:

1. current epoch's yield (baseYield);
2. previous epoch's boosted yield (boostedYield).

However, current epoch's yield may be transferred to the `tradeExecutor`, after the current share price was calculated. When this happens, the transferred yield won't be included in the total vault funds (i.e. vaults balances will be reduced by the amount of yield transferred). And since boosting may result in a smaller or a bigger amount of yield, it's not known how much yield is deposited into vault until boosted yield is withdrawn from `tradeExecutor` (which happens in another epoch).

Recommended Mitigation Steps

Consider transferring `baseYield` to the `tradeExecutor` before receiving `boostedYield` and before calculating the current share price. This will ensure that the current share price is calculated only after finalized yield (boosted yield from a previous epoch or the current epoch's yield that's not transferred to the `tradeExecutor`) is deposited into the vault.

[Medium-03] `Vault` share price manipulation may result in stolen deposits

> ### Resolution

> As per the recommendation, the team added a minting of at least 1e3 shares during the initialization of the `Vault` contract.

Targets

- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Vault.sol#L131-L133

Impact

When the vault has low liquidity, a malicious actor can manipulate the price of 1 share and steal deposits from future depositors.

Proof of Concept

The Vault is vulnerable to the classical inflation attack: when the vault has low liquidity, a malicious actor may inflate the price of 1 share by depositing funds directly; users who deposit next will get a reduced amount of shares due to the rounding during the division operation in the asset-to-shares conversion. Consider this exploit scenario:

1. The vault is empty, `lastEpochYield` is 0.
2. Alice deposits 1 wei of the asset and receives 1 wei of shares.
3. Alice sends `1000e18 - 1` of the asset directly to the vault, inflating the price of the share: the 1 wei of share can now be redeemed for `1000e18` of the asset.
4. Bob deposits `1999e18` into the vault and receives only 1 wei of shares, due to rounding: `totalSupply * amountIn / totalVaultFunds = 1 * 1999e18 / 1000e18 = 1`.
5. Alice redeems her 1 wei of shares and receives: `sharesIn * balance of the vault / totalSupply = 1 * 2999e18 / 2 = 1499.5e18` of the asset.

Taking into account the need to pay `lastEpochYield` when depositing, the exit fee when withdrawing, and the current liquidity of the pool, the cost of the attack may increase and the magnitude of the inflation may reduce, but the attack vector still remains.

Recommended Mitigation Steps

Consider sending first 1000 wei of shares to the zero address: this will increase the cost of the attack when the vault is empty. However, this won't mitigate the attack entirely as it doesn't fix the cause of the attack: rounding during share price calculation. Alternatively, consider increasing the number of decimals for shares: this will mitigate the rounding error, however the number of decimals of the `Vault` contract (which implements ERC20) will need to be increased, which may cause issues with integrating the `Vault` (as an ERC20 token) with third-party protocols and services. For more details, consider this discussions:

1. Implement or recommend mitigations for ERC4626 inflation attacks.
2. Address EIP-4626 inflation attacks with virtual shares and assets .

[Medium-04] Lack of transaction expiration check when selling LDO during harvesting

> ### Resolution
>
> The issue was resolved by adding a `uint256` deadline field to the `HarvestParams` structure and passing it to the Uniswap's swap function.

Targets

- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Harvester.sol#L203

Impact

Positive slippage may be stolen by MEV bots when harvesting yields.

Proof of Concept

When harvesting yields, the reward tokens (`BAL` and `LDO`) are swapped for `WMATIC` : LDO is swapped on Uniswap and BAL is swapped on Balancer. Each of these operations has a slippage protection, however their implementations are different:

1. the price that's used to calculate the expected amount of `WMATIC` when selling `LDO` is specified by the called (Harvester.sol#L191-L193), i.e. it's computed before the transaction is set;
2. the price that's used to calculate the expected amount of `WMATIC` when selling `BAL` is computed on-chain, by querying Chainlink oracles (Harvester.sol#L210-L216), i.e. it's computed when the transaction is executed.

The swapping functions of the exchanges take the deadline parameter: the timestamp after which the transaction is considered expired (Swaps.sol#L74, UniswapV2Router02.sol#L230). However, the parameter's value is set to `block.timestamp`, which disabled the transaction expiration check. This is not critical for the Balancer swap because its slippage tolerance is computed on-chain, when the transaction is executed. However, the Uniswap swap can be impacted.

Example exploit scenario:

1. The keeper executes a transaction calling `Vault.harvest()`. The keeper sets HarvestParams, which only includes `ldoPriceInMatic`. The transaction withdraws `LDO` and `BAL` rewards from the Curve and Balancer gauges and sells them for `WMATIC`.
2. Mining of the transaction is delayed due to a sudden spike of the cost of gas. While the transaction is delayed, the price of `LDO` increases, which should result in more `WMATIC` after the swap.
3. Since the expiration check is disabled in the swap call, the transaction won't expire. However, the positive slippage that comes from the increased price of `LDO` will be stolen by a MEV bot via a sandwiching attack.
4. The slippage tolerance check won't protect against the attack since `ldoPriceInMatic` was computed before the transaction was executed, thus the expected output amount of `WMATIC` will be computed based on the outdated price.

Recommended Mitigation Steps

When swapping `LDO` for `WMATIC`, consider using price oracles to calculate the price of `LDO` in terms of `WMATIC`, similarly to how it's done when swapping `BAL` for `MATIC`. Or consider setting the deadline parameter of the `swapExactTokensForTokens` call to a reasonable timestamp, after which a harvesting transaction should be considered expired (for example, Uniswap sets it to 30 minutes for Polygon).

[Medium-05] Missing Chainlink round completeness check

> ### Resolution
>
> Resolved by adding the `answeredInRound >= roundId` requirement to the validateRound function.

Targets

- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Harvester.sol#L257
- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Harvester.sol#L262
- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Harvester.sol#L278-L287

Impact

Pricing `BAL` in terms of `WMATIC` may be impacted by an outage in Chainlink oracles, resulting in a wrong slippage tolerance value.

Proof of Concept

As per the documentation of Chainlink:

> You can check answeredInRound against the current roundId. If answeredInRound is less than roundId, the answer is being carried over. If answeredInRound is equal to roundId, then the answer is fresh. However, a round can time out if it doesn't reach

> consensus. Technically, that is a timed out round that carries over the answer from the previous round.

However, there's no round completeness checks in the `_getPriceOfAmountAinB` function of `Harvester`.

Recommended Mitigation Steps

In the `validateRound` function, consider checking that `answeredInRound` equals to `roundId`, as returned from the `latestRoundData` calls.

[Low-01] Unsafe transferring of ERC20 tokens in `sweep` functions

Targets

- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Vault.sol#L567-L571
- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/BalancerYieldExecutor.sol#L155-L159
- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/CurveYieldExecutor.sol#L152-L156

Impact

Governance may not be able to sweep some ERC20 tokens that were mistakenly sent to `Vault`, `BalancerYieldExecutor`, and `CurveYieldExecutor`.

Proof of Concept

The `Vault`, `BalancerYieldExecutor`, and `CurveYieldExecutor` contracts implement `sweep` functions that allow the caller to transfer any ERC20 token from the contract to the governance. The functions call the `transfer` method on the specified address and expect that the call returns `true`. However, not all ERC20 implementations follow the standard, and some of them don't return a boolean in the `transfer` method (one example is USDT token on the Ethereum mainnet). As a result, trying to transfer such tokens from the contracts will always result in a revert.

Recommended Mitigation Steps

Consider using the SafeERC20 library to handle token transfers in the `sweep` functions. This is only recommended for the `sweep` function since they're intended to be used with an arbitrary ERC20 token.

[Low-02] Discrepancy between `sweep` function's code and documentation

> ### Resolution
>
> Response from the team: "The `sweep` behaviour locked to onlyGovernance is the expected behaviour." However, the documentation of the functions wasn't updated.

Targets

- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Vault.sol#L566
- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/BalancerYieldExecutor.sol#L154
- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/CurveYieldExecutor.sol#L151

Impact

Sweeping tokens from `Vault`, `BalancerYieldExecutor`, `CurveYieldExecutor` may be impacted if the intended implementation is to allow anyone to call the `sweep` function.

Proof of Concept

The `sweep` function of the `Vault`, `BalancerYieldExecutor`, and `CurveYieldExecutor` contracts can only be called by the governance, due to the `onlyGovernance` check. However, the documentation of the functions says:

> can be called by anyone to send funds to governance

If the intended implementation is to allow anyone to sweep tokens to send them to governance, then this won't be possible due to the `onlyGovernance` check.

Recommended Mitigation Steps

Consider updating the documentation of the functions or removing the `onlyGovernance` check.

[Low-03] Harvesting may be impacted by low liquidity in QuickSwap and Balancer pools

Targets

- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Harvester.sol#L59-L61
- https://github.com/Brahma-fi/matic_vault/blob/c88fc4bafabe2f59fde284afae58ac16eb67fc79/src/Harvester.sol#L64-L65

Impact

If liquidity moves from the hardcoded QuickSwap and Balancer pools that are used to swap LDO and BAL for WMATIC, the swaps may be affected by increased slippage, resulting in lower swap prices and reduced yields.

Proof of Concept

The `Harvester` contract hardcodes the QuickSwap router address and the Balancer pool address, not allowing to migrate to pools with higher liquidity and lower slippage. If liquidity is migrated from one of the hardcoded pools (the QuickSwap router will always use the same pool for the LDO/WMATIC pair), selling harvested yields may happen at a worse price due to increased slippage. In the worst situation, swapping may not be possible at all if all liquidity is removed from one of the pools.

Recommended Mitigation Steps

Consider implementing a solution that will allow the protocol to use different pools to sell yield without re-deploying the contract. One solution is a "swapper" contract that implements swapping logic using one or multiple pools and that can be changed by governance.