Brahmafi: Polygains v2 Code Review

Conducted by Bluethroat Labs

Auditor(s): Rahul Saxena (twitter.com/saxenism)

Parth Patel (twitter.com/__parthpatel__)

February 3rd, 2023
Patch Update - 1: February 9th, 2023

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held understanding of the auditor's knowledge of security patterns as they relate to the client's contract(s), assuming that blockchain technologies, in particular, will continue to undergo frequent and ongoing development and therefore introduce unknown technical risks and flaws. The scope of the audit presented here is limited to the issues identified in the preliminary section and discussed in more detail in subsequent sections. The audit report does not address or provide opinions on any security aspects of the Solidity compiler, the tools used in the development of the contracts or the blockchain technologies themselves, or any issues not specifically addressed in this audit report.

The audit report makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, the legal framework for the business model, or any other statements about the suitability of the contracts for a particular purpose, or their bug-free status.

To the full extent permissible by applicable law, the auditors disclaim all warranties, express or implied. The information in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. The auditors hereby disclaim, and each client or user of this audit report hereby waives, releases and holds all auditors harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

Hash of the codebase being audited **c88fc4bafabe2f59fde284afae58ac16eb67fc79**

Issues Found

1. Vault.sol: First pool depositor can be front-run and part of their deposits can be stolen

Category: High **Current Status: Fixed**

Context: Vault.sol deposit() function

Description: The first deposit with a totalSupply of zero shares will mint shares equal to the deposited amount. This makes it possible to deposit the smallest unit of a token and profit off a rounding issue in the computation for the minted shares of the next depositor

Example:

- The first depositor (victim) wants to deposit 2M(2e24) Matic and submits the transaction.
- The attacker front runs the victim's transaction by calling deposit(1) to get 1 share.
- They then transfer 1M Matic (1e24) to the contract, such that totalVaultFunds = 1e24 + 1, totalSupply = 1.
- When the victim's transaction is mined, they receive 2e24 / (1e24 + 1) * totalSupply = 1 shares (rounded down from 1.9999...).
- The attacker withdraws their 1 share and gets 3M Matic * 1 / 2 = 1.5M Matic, making a 0.5M profit.

Recommendation:

Require a minimum initial shares amount for the first deposit by adjusting the initial mint (when totalSupply == 0) such that:
• mint an INITIAL_BURN_AMOUNT of shares to a dead address like address zero.

• only mint depositAmount - INITIAL_BURN_AMOUNT to the recipient. INITIAL_BURN_AMOUNT needs to be chosen large enough such that the cost of the attack is not profitable for the attacker when minting low share amounts but low enough that not too many shares are stolen from the user. We recommend using an INITIAL_BURN_AMOUNT of 1000.

2. Vault.sol: Wrong issuance of shares to the depositors

Category: High **Current Status: Fixed**

Context: Vault.sol deposit() function

Description: The calculation of shares given to the depositors who deposits funds via deposit() function results in incorrect awardance of shares to depositors.

Example:

- The first depositor deposits 1000 Matic and get 1000 shares.
- The second depositor also deposits 1000 Matic. Ideally, second depositor should also get 1000 shares but he will get 500 shares.

Recommendation: Track balance of vault before deposit by keeping separate tracking variable which can be used to know the balance of vaults before calling deposit.

3.Vault.sol: Withdraw by user is not bounded by 1 week timeline

Category: High

Context: Vault.sol withdraw() function

Description: The assumption(as discussed with team) is that the withdraw of vault tokens will only be initiated after atleast one week of deposit. But there is no such condition enforced in smart contract. This can lead to loss of funds by the users interacting directly with smart contract.

Let's say I deposit 1000 wantTokens in the vault. I will get shares of it. Immediately, I try to withdraw that tokens by giving shares to vault. In this case, I won't get 1000 wantTokens but less than that. Let's say I got 950. Then there is no accounting for remaining 50 wantTokens

Example:

- Let's assume user deposit 1000 wantToken in the vault. Let's assume lastEpochYield is x when he deposits and x is positive integer.
- The shares that user will get when he deposits is (totalSupply() * amountIn) / (totalVaultFunds() + lastEpochYield)
- Now, if user doesn't wait for 1 week and tries to withdraw before that, the wantToken he will get is (sharesIn * IERC20(wantToken).balanceOf(address(this))) / totalSupply().
- Putting value of sharesIn that we got in above formula, we will get wantToken which is equal to (amountIn * IERC20(wantToken).balanceOf(address(this)))/(totalVaultFunds() + lastEpochYield)
- So, the value of amountOut is less than user deposited amount. These funds won't get accounted for by anyone and can't be claimed.

Recommendation:

These require changes in design and approach the team is taking to issue and withdraw shares.

4. Vault.sol: Attacker can grief and take most of the other user's reward in first epoch

---

Category: High

Context: Vault.sol deposit() and withdraw() function

Description: The attacker can just deposit a large amount before harvest in first-week cycle(when lastEpochYield = 0). In that case, the user can accumulate a large number of shares and can immediately withdraw after harvest is called collecting rewards in proportion to the shares he got earlier. If the shares are larger, the attacker can just claim other users' rewards by depositing for a very short time.

Recommendation: These require changes in design and approach the team is taking to issue and withdraw shares.

5. Vault.sol: Lack of zero checks in constructor

Category: Medium **Current Status:** Fixed

Context: Vault.sol constructor

Description: State variables set in the constructor lacks a zero check. Also, some of the variables are not supposed to change, so can be set as immutable.

Recommendation: Set up sanity checks everywhere any kind of user input is being taken.

6. Hardcoded Slippage Tolerance

Category: Medium

Context: CurvePositionHandler and BalancerPositionHandler

Description: For both the `CurvePositionHandler` and the `BalancerPositionHandler` the slippage is set as a constant value and can only be changed by the protocol owners `onlyGovernance()`. The issue here is more on the side of user experience as opposed to security issues. Ideally, we would like to give the users the option to set their own slippage tolerance and then execute any strategy.

This becomes even more important when a user wants to engage with a huge amount and wants to minimize the slippage for fear of loss due to sandwich/MEV attacks.

Recommendation: Consider making the slippage user dependent.

7. Hardcoded External Contract Addresses

Category: Medium

Context: Harvester.sol

Description: Consider the following lines of code where the addresses of external contracts are being set.

```
IBasePool public constant override wMaticBalPool =
IBasePool(0x0297e37f1873D2DAb4487Aa67cD56B58E2F27875);
```

Here, we are setting the addresses of external contract as constants, so in the case of these external contracts getting upgraded, we do not have the option of updating `wMaticBalPool` therefore rendering it useless

Recommendation: Avoid hardcoding the addresses of external contracts into your contract as they can be changed/upgraded. For example: `IAggregatorV3 public constant override maticUsdPriceFeed = IAggregatorV3(0xAB594600376Ec9fD91F8e885dADF0CE036862dE0);`. Instead, try and pass these addresses using the constructors and make sure they are not `constants` so that they can be updated/upgraded by `onlyGovernance` in case their addresses get changed.

General Recommendations

This section consists of general recommendations given to the protocol's maintainers based on the contracts' overall design and coding philosophy.

These are not necessarily security flaws and are dependent completely on the maintainers' discretion for their implementation.

1. Consider using custom errors instead of `require` strings to save gas—introduction blog.
2. For simple mathematical operations that are guaranteed not to overflow, consider using `unchecked` blocks.
3. Do not initialize variables that are being initialized now with their current value. For example: `uint256 baseYield = 0;` can be changed to `uint256 baseYield;`
4. Pre-increment operators are cheaper than post-increment operators
5. Do not read values for length inside of a loop to save gas. For example: `for (uint256 i = 0; i < totalYieldExecutors(); i++)` can be changed to

   ```
   uint256 totalYieldEx = totalYieldExecutors();

   for(uint256 i; i < totalYieldEx; ++i)
   ```

6. Consider changing the `public` functions which aren't called anywhere else inside the contracts to `external` to save on gas costs.
7. Consider adding sanity checks for all user inputs in user-facing functions. For example, in the function `Harvester.harvest()`, a user can easily pass `bytes("")` as `_data`.
8. Make sure the compiler versions are consistent across all files. For example, in `Harvester.sol` the solc version being used is `>= 0.7.6` which means that there is no inherent overflow/underflow check. Also, in this contract, the `safeMath` library has not been used so no mathematical operation in this contract is safe.

Protocol/Logic Review

Part of our audits are also analyses of the protocol and its logic. The **Bluethroat Labs** team went through the implementation.

Partial tests and no documentation has been provided to us and although it was difficult to understand the goal of the protocol, we managed to do so from the functions, inline documentation, and prompt responses from the protocol developers.

According to our analysis, the protocol and logic are working as intended, given that the findings listed in the *Issues* section are fixed, however, we **strongly recommend** another **follow-up audit** post implementing the suggestions listed here. **And**, the next audit should be preceded with **a thorough and comprehensive testing suite** for the protocol. Emphasis should be laid on much more detailed end-to-end fuzzed tests along with integration tests.

We were not able to discover any additional problems in the protocol implemented in the smart contract.