Balancer

MultiRewards and StablePool

OpenZeppelin | security

Introduction

The Balancer team asked us to review and audit updates to their protocol smart contracts. This is our third engagement with Balancer, where we previously audited both the v1 and v2 of their protocol. We looked at this new code and now publish our results.

We audited commit `d430cdbb15468bc6f4c778f6268d98f36648ebf4` of the `balancer-labs/balancer-v2-monorepo` repository.

In scope were the following contracts:

`pkg/distributors/contracts/MultiRewards.sol`
`pkg/pool-stable/contracts/StablePool.sol`
`pkg/pool-stable/contracts/StableMath.sol`

Overall health

We found the Balancer team to be highly responsive, pleasant to work with, and eager to incorporate feedback.

We are always quite impressed with Balancer's elegant and faithful implementation rooted in solid and sound mathematical foundations, showcasing a real mastery of the Solidity programming language and system design.

While we did audit their v2, it is worth noting that aside from the files within this audit scope, their codebase has undergone a refactor to conform to the Yarn2 monorepo structure as well as other improvements. Even if changes to certain modules weren't drastic, we encourage a full audit of the changes to ensure no bugs were introduced.

High level overview of changes

- Adding `MultiRewards` staking distributor contract
- Updated iteration of the `StablePool` AMM
- Updated `StableMath` contract to support new `StablePool`

`MultiRewards` staking distributor contract

The `MultiRewards` contract provides an opportunity for users to earn rewards in a reward token by staking a pool token. This design is a mutation of the Curve Finance `MultiRewards` contract.

The `MultiRewards` contracts both allow for multiple different tokens to be distributed to a liquidity providers for different pools. Balancer's implementation has added a whitelisting functionality, so that

only approved "rewarders" can set up disbursements of rewards. Additionally, functions exist to claim rewards "internally", without transferring them out of the Balancer system, and to stake balancer pool tokens on behalf of another user by using `ERC20Permit`.

It is the intent that the rewarders will be Balancer governance distributing BAL, asset managers distributing tokens earned through asset management, and other `allowlisted` users powering airdrops to liquidity providers.

There is an added functionality allowing the user to claim rewards to a callback contract. Note that the `exitWithCallback` function is left as reentrant to enable a callback contract to immediately join and stake for another pool.

`StablePool` AMM

v2 of Balancer builds on top of their previous automated market maker design allowing more efficient capital allocation, cheaper gas prices and the usage of unused liquidity by asset managers creating an innovative DeFi primitive. This v2 design has as its hub a `Vault` contract, which is the holder of all the assets of the protocol. Plugged into the `Vault` are `Pool`'s, AMMs which can create their own trading algorithms, giving the `Vault` more capabilities without the need to redeploy nor migrate funds from the `Vault`.

This `StablePool` is one such AMM to be plugged into the `Vault`. The `StablePool` is intended to be a cross-market for stable-coins with very minimal price slippage on the demand side, as well as a multi-stablecoin "savings account" yielding high returns on the supply side.

`StableMath` contract

The `StableMath` contract is in a way the "engine" for the `StablePool` since pool joins, exits, and swaps are governed by an invariant that is defined mathematically. The invariant is defined by a relation combining the linear constant-price and constant-product invariants in a way that the invariant is relatively stable when the pool is near balanced, but shifts towards the constant-product invariant as the pool's holdings becomes more imbalanced.

This invariant, along with inputs/outputs to swaps, are calculated within the `StableMath` contract. Ultimately, calculations of all of these quantities rely on approximations efficiently computed using the Newton-Raphson method.

Privileged roles

`Pools` can designate asset managers who will be able to withdraw funds from the pool to use the underlying liquidity, so that liquidity is not dormant in the vault. This is extremely useful, but it also raises security concerns since asset managers can realize both profit and losses to the liquidity providers. As such, asset managers are meant to be assigned to smart contracts to remain trustless, since converting an externally owned account to an asset manager will allow this account's owner to be able to use those assets as she/he wants.

The `MultiRewards` contract has a privileged group of `allowlisters` which is comprised of pool asset managers, the pool's themselves, or other accounts that the `Authorizer` allows. These `allowlisters` are the only entities allowed to call `allowlistRewarder` which is the way to register members of another privileged group: the `allowListedRewarder`'s. The `allowListedRewarder`'s are the only entities allowed to call `addReward`, which creates a new reward disbursement within the `MultiRewards` contract.

The `StablePool` is initialized without asset managers, so the trust model is simplified, although it is initialized with an `owner`, whose elevated `_canPerform` status gives it no extra abilities in the pool.

Trust Assumptions

There are a few assumptions which the system relies on for safe operation. Ultimately, a user must trust that privileged roles uphold these assumptions. The Balancer team is aware of these assumptions and have designed the system to contain any risk created by violating these assumptions. Users should note that:

- Rewards tokens which rebase or charge fees on transfers may result in unclaimable rewards due to the contract attempting to transfer more tokens than it has.
- Pools which have too many rewards tokens may be unable to disburse rewards due to running out of gas in rewards calculations.
- There is no way to remove a rewarder once they are approved for a given pool. This means that a pool is at risk of any rewarder becoming malicious at any time, possibly by a malicious entity gaining control of their private keys.

Generally speaking, these risks are contained to only affecting the pools which they involve. So a malicious rewarder, for example, can only negatively affect the pools which they are approved as rewarders for. An exception is if multiple pools have rewards tokens which rebase or charge fees, then they all may share some risk due to rewards tokens being held in the same contract. For instance, one pool may exhaust the available balance of rewards tokens, causing other pools to be unable to disburse rewards.

Findings

Here we present our findings

Critical severity

[C01] Rewards rate can be reduced by an enormous factor

The `MultiRewards` contract is a hub where rewarders can post funds in a reward token to reward users who stake their pool token. There can be many `rewarder`'s to a `pool` `rewardToken` combo, and to each there is an associated `rewardRate`. So that for a given `pool` `rewardToken` combo, the reward the users earn is the accumulation of rewards, each associated to these `rewarder`'s `rewardRate`'s.

The `rewardRate` is established when the `rewarder` calls the `notifyRewardAmount` function where they deposit their `reward` in `rewardToken`. Note that the `notifyRewardAmount` call could actually be made by the `RewardsScheduler` on behalf of the `rewarder`.

There are two scenarios to consider. The first is the `rewarder` making this call outside of the reward period. In this first scenario, the `rewardRate` is simply the quotient of the deposited `reward` and `rewardsDuration`. In the second scenario, where this call is made within the reward period, the `rewardRate` is computed taking into consideration the former `rewardRate` over the time that has already lapsed. Indeed the arithmetic in this section is more complicated.

The problem is that the implementation of the arithmetic corresponding to this second scenario has a bug where the `rewardRate` can be set to be off by almost a factor of `1e18`.

This is due to the `leftover` quantity being the product of `remaining` and the former `rewardRate` using the `FixedPoint` `mulDown` multiplication operation. To see this, we note the `FixedPoint.mulDown` operation is designed to ingest factors each in terms of `FixedPoint.ONE` (`1e18`) and returns their product dividing out the "extra" `ONE` factor. But the `remaining` factor is **not** in terms of `ONE` while the `rewardRate` is. This means that the `leftover` value is off by a factor of `1e18`.

Since the resulting `rewardRate` in the second scenario has numerator being the sum of `reward` and `leftover` this can greatly impact its value. In the limit case it can approach being off by a factor of `1e18`.

The effect of this bug is that users' rewards can be drastically reduced anytime `rewarder`'s supplement the `reward` by way of `notifyRewardAmount`.

Consider fixing this bug by using the `Math.mulDown` operation instead to calculate the `leftover` variable.

High severity

[H01] `setRewardsDuration` may accidentally lock out a rewarder

The `setRewardsDuration` function of the `MultiRewards` contract updates the `rewardsDuration` for a given `pool` `rewarder` `rewardsToken` combination. This function validates that the reward period is still

active and that the new `rewardsDuration` is non-zero. The `rewarder` in this context is the `msg.sender`.

A `msg.sender`, for which there is no active `rewardData` added, can call `setRewardsDuration` with non-zero `rewardsDuration` since the check that the period is still active only checks that the `block.timestamp` is after the `periodFinish` of the `rewardData` for the `pool` `rewarder` `rewardsToken` combo of this call. Under these circumstances, this comparison will always pass since the `periodFinish` has never been set and is thus zero.

An effect of setting this value is that a potential rewarder can accidentally lock themselves out of adding rewards if they were to mistakenly call `setRewardsDuration` before they were to `addReward`. This is because we see that `setRewardsDuration` would set the `rewardsDuration` for such a `msg.sender` to a non-zero value, but the `addReward` function requires that the `rewardData` for its `pool` `rewarder` `rewardsToken` is zero.

Consider adding `onlyAllowlistedRewarder` modifier to this setter, and also adding safeguards to enforce `setRewardsDuration` can only be called after a rewarder calls `addReward`.

Medium severity

[M01] ERC20 `permit` incorrectly used

The use of ERC20Permit within the `stakeWithPermit` function uses `msg.sender` as the owner of the funds. Note that within `ERC20Permit`, within the `permit` function, the `owner` is the first variable, which is checked to be the `signer` of the signature presented. As can be seen in the call to `permit`, the first parameter is `msg.sender`.

The result is that this call to `permit` will effectively be an overcomplicated way to call "approve". If it is intended that some user to be able to stake the tokens of some other user, then the first parameter in the call to `permit` will need to be changed from `msg.sender` to a user-controlled input. Fortunately, as the call exists now, it only allows a user to approve their own tokens, so this does not pose a danger to other user's tokens.

Consider changing the call to `permit` such that the first parameter is some user-controlled input rather than `msg.sender`. This will allow `permit` to function as intended.

[M02] Errors and omissions in events

Throughout the codebase, events are used to signify when sensitive actions are performed. However some events are missing important parameters. Some events are emitted in a disorderly and confusing manner following certain sensitive actions. Additionally, some sensitive actions are lacking events altogether.

Events missing important parameters include:

- The `RewardsDurationUpdated` event of the `MultiRewards` contract has entries for `pool`, `token`, and `newDuration` but does not have an entry for `rewarder`. Since the `rewardData` is dependent on `pool`, `rewarder` and `token`, there can be different `rewardData` for different `rewarder`'s, each having their own durations.

- The `RewardAdded` event which is called in the `MultiRewards` contract has parameters for `rewardToken` and `reward` but does not indicate which `pool` it is being added to.

Confusing events include:

- The `AmpUpdateStopped` and `AmpUpdateStarted` events. Owing to the fact that one version of `_setAmplificationData` calls the other version of `_setAmplificationData`, both events will be emitted in a single transaction when the `constructor` of `StablePool` calls `_setAmplificationData`. Additionally, if `startAmplificationParameterUpdate` is called followed by a call to `stopAmplificationParameterUpdate`, two instances of `AmpUpdateStarted` will be emitted followed by one instance of `AmpUpdateStopped`. Finally, when an amplification parameter update is successfully completed, no event will be emitted to indicate this, leaving the chain of events including only one `AmpUpdateStarted` event.

Sensitive actions that are lacking events include:

- The `allowlistRewarder` function of the `MultiRewards` contract, approves an account to be within the `_allowlist`, but does not emit an event signifying this change in privileges.

Consider making all of the above changes to enable off-chain clients to correctly track sensitive actions in the Balancer protocol.

[M03] `getRewardForDuration` can return incorrect values

The `getRewardForDuration` function of the `MultiRewards` contract returns the product of the `rewardRate` and the `rewardsDuration` for a given `pool` `rewarder` `rewardsToken` combo. The `rewardsDuration` for a reward is initially set at `addReward` and is used to calculate the `rewardRate`. After the `periodFinish`, the rewarder can update the `rewardDuration` by way of the `setRewardsDuration` function.

The problem is that the `setRewardsDuration` function does not also update the `rewardRate`. This way, the product given by `getRewardForDuration` will be between unrelated factors, and will thus produce an incorrect value.

Consider refactoring to ensure that the product returned by the `getRewardForDuration` function accurately reflects the true reward rate.

Low severity

[L01] Error prone math

In line 489 of `StableMath.sol`, a multiplication happens after a division step in a calculation.

If the order of these two operations is swapped, a slightly more accurate answer can be obtained owing to the loss of information due to a division happening later, rather than earlier, in the calculation. If the division is the final step, the result in solidity will be closer to the mathematically correct answer, which may have a decimal component.

Consider rearranging the `mul` and `divDown` operations for better accuracy in this calculation.

[L02] Insufficient error documentation

Currently, in the balancer docs there exists an "error codes" page. However, many of these error codes do not provide sufficient information for a user to correct the error-causing behavior. For many of these error codes, the user would need to search through the balancer codebase and trace the error, which may be very difficult for users with less technical background. For example, the following error codes do not provide enough information on their own to be corrected.

- `206 UNINITIALIZED`
- `424 RENOUNCE_SENDER_NOT_ALLOWED`
- `517 UNALLOCATED_ETH`

Consider providing a page in the documentation for more explicit explanations of each error code. The explanations should be detailed enough that the user can correct the error-causing behavior.

[L03] Misleading error messages

Several error messages in require statements were found to be too generic, not accurately notifying users of the actual failing condition causing the transaction to revert. In particular:

- Line 123 of `MultiRewards.sol` checks the `rewardsDuration`, but the error message makes no reference to this specific parameter.
- Line 409 of `MultiRewards.sol` does not clearly indicate that `msg.sender` must be the `rewarder` or `rewardsScheduler`.
- Line 239 of `StablePool.sol` simply returns the error code `BAL#206` which codes for the error `UNINITIALIZED`. This should instead notify the user that they have set their `JoinKind` incorrectly.

Error messages are intended to notify users about failing conditions, and should provide enough information so that the appropriate corrections needed to interact with the system can be applied. Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality. Therefore, consider not only fixing the specific issues mentioned, but also reviewing the entire codebase to make sure every error message is informative and user-friendly enough.

[L04] Naming issues

Within the codebase, there are several instances of misleading names for functions or values. For example:

- Within the `MultiRewards` contract, the `getRewardForDuration` function should indicate that it does not transfer any funds. The similarly named `getReward`, `getRewardAsInternalBalance`, and `getRewardWithCallback` functions all transfer accrued rewards.

- The variables called `remaining` and `leftover` may benefit from being renamed to `remainingTime` and `leftoverRewards`, to help differentiate the meaning of the two.

- Within the `StablePool` contract, the name `balances` is used often, sometimes referring to balances in terms of underlying token units, and sometimes referring to balances scaled to `1e18` units per token. For example, line 142 and line 171 are storing "scaled" balances, while line 593 is storing "unscaled" balances. Consider renaming instances of `balances` to `scaledBalances` and `rawBalances` or `unscaledBalances` for greater clarity.

- Throughout the `StableMath` contract, the `amplificationParameter` name is often used to refer to a value which is `A*n^(n-1)`. For example, this is done on line 50 and line 110. This may confuse developers who believe that it simply refers `A` as it is used in the StableSwap paper. Consider renaming this variable to better differentiate between it and the "amplification coefficient".

- On line 266 and line 365 of `StableMath.sol`, the variable `newBalanceTokenIndex` is ambiguously named. It stores a balance, but based on its name a reader may assume it is storing an index. Consider renaming to `newBalanceAtTokenIndex`

- On lines 221, 228, and 233 of `StableMath.sol`, `amountInWithoutFee` should be `amountInWithFee`.

- On line 287 of `MultiRewards.sol`, "added" would be clearer if it was instead "staked".

Since many of these issues affect the public API of the codebase, it is imperative that naming is clear and correct. If naming is ambiguous, serious complications may arise from developers misinterpreting the names of functions or values. Additionally, accurate naming increases the security of the project by making clearer the intent behind each function or value. Consider renaming the listed examples.

[L05] Not following NatSpec

The docstrings for many of the functions in the codebase are not following the Ethereum Natural Specification Format (NatSpec). For example:

- The function `allowlistRewarder`.
- The function `isAllowlistedRewarder`.
- The function `getLastInvariant`.

Note that this list is not exhaustive. Consider following this specification on every function that is part of the contracts' public API.

[L06] Using old Solidity version

Throughout the codebase, Solidity `^0.7.0` is used. However, at the time of writing many new versions of the Solidity compiler have been released, each having improvements in features including bug fixes. Consider upgrading the contracts in this codebase to use the latest version of Solidity, 0.8.7.

[L07] Redundant arithmetic

The `_calcDueTokenProtocolSwapFeeAmount` function of the `StableMath` contract returns the fee amount to pay in a given token. Its final return value is to be the product of locally computed `accumulatedTokenSwapFees` and its `protocolSwapFeePercentage` parameter.

But as implemented, there is an additional `.divDown(FixedPoint.ONE)` operation chained to this return value. Since the `divDown` function scales the numerator by `ONE` before performing the native division operation, this additional operation is redundant.

This redundancy can be seen algebraically as `a.divDown(ONE)` = `aInflated / ONE` = `a * ONE / ONE` = `a`.

The effect of this redundant operation is an additional gas cost to invoke this function, as well as cluttering the readability of an already dense, mission-critical, math-heavy section of code.

Consider removing this redundant division operation from the return value of this function.

[L08] Unhandled silent failure

The `_onInitializePool` function of the `StablePool` contract sets the `_lastInvariant` to its first non-zero `invariantAfterJoin` by way of calling the `_updateLastInvariant` function.

A user could naively try to exit an uninitialized pool by calling the `exitPool` function on the vault which has in its callstack a call to the `_onExitPool` hook. This `_onExitPool` hook makes a direct call to the

`_getDueProtocolFeeAmounts` function which uses the `_lastInvariant` which in this scenario is zero. This way there will be a revert in a division step whose denominator is this zero `invariant`.

This failure is not handled early nor routed to some explicit error message, it can unexpectedly stop the execution, reverting without any explicit reason.

Following the "fail early and loudly" principle, consider including specific and informative error-handling structures to avoid unexpected failures.

Notes & Additional Information

[N01] `exitWithCallback` directs only stake, not reward, to callback

The `exitWithCallback` function of the `MultiRewards` contract allows the unstaked token to be directed to the `callbackContract` so it can make use of the unstaked funds. However the rewards that are reaped within the same routine cannot be directed to the `callbackContract` in the same way but are directed only to the `msg.sender`.

There is not thorough supporting documentation making explicit to the users this subtlety in how the funds associated with this call will be directed.

Consider either updating the documentation to thoroughly explain how users expect their funds to be directed, or enhance the `exitWithCallback` function so that the user can direct both the unstaked funds and the rewards.

[N02] Opportunities for gas improvements

- Unnecessary intermediate variable `totalTokens`.
- L438 of `MultiRewards` can use native subtraction instead of `.sub` since in this block it is guaranteed `block.timestamp` < `.periodFinish`.
- L718 and L719 of `StablePool` the first two checks that `totalTokens` exceeds the index are redundant since we know `totalTokens` >= 2.
- `sumBalances` is computed for many routines within the `StableMath` contract, but it may be more efficient to store this value within the state of a pool and update it as balances change.
- L211 and L315 of `StableMath` the respective computations of `invariantRatioWith*Fees` could be more efficient by multiplying the reciprocal of `sumBalances` with the sum of the combined balances and in/out amounts. This is because the `currentWeight` includes a multiplication by `balances[i]`, while the `balanceRatiosWith*Fee` includes a division by `balances[i]`. These cancel out. Additionally, the step of dividing by `sumBalances` in `currentWeight` can be done once at the end of the calculation, rather than for each token in the pool, since `sumBalances` does not change.

- The `_getReward` function of the `MultiRewards` contract has "ops" ran for every `pool`/`rewardToken` combo regardless of whether the `reward`>0. Each of these ops has a corresponding event emitted, so these trivial ops can not only waste gas but can add to the noise for off-chain clients processing such events.

[N03] Incorrect function visibility

There are some occurrences of functions being marked as `public`, while they should be defined as `external` because they are not called anywhere internally.

- The `isAllowlistedRewarder` function of the `MultiRewards` contract.
- The `totalEarned` function of the `MultiRewards` contract.
- The `getRate` function of the `StablePool` contract.

Consider changing these functions to be defined as `external` to better align with the principle of least privilege, and to reap the benefits of possible improvements in gas performance.

[N04] Comment should be moved

Consider moving the comment on lines 595-6 of `StablePool.sol` to above line 601 to clarify the intent of the comment.

[N05] Lack of documentation regarding "taxable amounts"

Within `StableMath.sol`, there are a few instances of calculations involving "taxable" or "non-taxable" amounts. For example, around line 225 and around line 284. The application of a "tax" is a somewhat confusing feature, and as it is common in the `StableMath` contract it should be documented more completely.

Consider creating some external documentation to explain the intention behind the "taxable" amount calculation. Additionally, consider adding inline documentation, possibly linking to the external documentation within the functions that apply taxes. By creating documentation, code reviewers and developers in the future will be able to better understand the intention of the code, and more easily build off of it. It will also become easier to spot errors or inconsistencies in the code if the intention is well defined.

[N06] Inconsistent style

There are various occurrences of style inconsistency within this codebase:

- `rewardToken` is used in comments but then `reward token` is used to mean the same thing.

- The `StablePool` contract inherits the `StableMath` contract, yet when it calls its methods, it calls them on the contract handle. This is unnecessarily verbose, since these methods are accessible directly.

- Similarly, the `StablePool` contract appends its own contract handle to enums on L238 and L239 within its own contract logic. This is unnecessary since they are accessible without this handle. Do note that this syntax is actually necessary on L649 and L650 to access the pointers of external functions.

- Calls to `InputHelpers.ensureInputLengthMatch` method within `StablePool` in some cases have the parameters ordered so that the target length is the first parameter, while in other instances such ordering is reversed.

- In `StablePool` contract, there is use of convenience functions `_isToken0(...)` and `_isToken1(...)` for the first two tokens, but not for the others.

- The `MultiRewards` contract uses `require` statements for errors, while the rest of the codebase uses the special `_require` function with accompanying error codes (for example, in `StablePool`).

Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style is recommended. Consider resolving the inconsistencies mentioned above.

[N07] Typos

We have identified the following typos in the code:

- On line 61 of `StableMath.sol`, the string of asterisks contains an "x", which should be removed.

- On line 210 of `StableMath.sol`, "without" should be "with".

- On line 484 of `StableMath.sol`, "fromm" should be "from".

- On line 38 of `StablePool.sol`, "multiple" should be "multiply".

- On line 667 of `StablePool.sol`, "alawys" should be "always".

- On line 678 of `StablePool.sol`, "paramater" should be "parameter".

Consider correcting the typos as suggested.

[N08] Unnecessary Import

In the `MultiRewards` contract, consider removing the import statement for `openzeppelin/SafeMath.sol`, as this contract is never used in `MultiRewards`.

Conclusions

1 Critical and 1 High severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface.