Introduction

The Balancer team asked us to review and audit the new version of the protocol's smart contracts. We looked at the code and now publish our results.

System overview

The new Balancer release builds on top of their previous automated market maker design and creates a new system which allows more efficient capital allocation, cheaper gas prices and the usage of unused liquidity by asset managers creating an innovative DeFi primitive. The system consists of a central `Vault` contract that holds the assets of all `Pools` in the system and allow users to interact with the liquidity in every `Pool` registered in the `Vault`. This design decision allows simpler user interactions when doing swaps across different `Pools` without compromising security as the `Vault` keeps track of the `Pool` balances isolated from each other. The `Vault`, however, does not makes assumptions on how each `Pool` should work, but rather use hooks to call `Pool` functions that will execute on swaps giving flexibility to `Pool` developers to build their own swap algorithms. As such, the users interacting with the `Vault` should trust the underlying `Pool` they are using to provide liquidity or perform swaps.

Vault

Balancer's new version presents a couple of design decisions in order to make swaps simpler and cheaper in terms of gas consumption. The most important one is the creation of a `Vault` contract, which is the holder of all the assets of the protocol. The `Vault` keeps track of the balance of each `Pool` isolated from each other, and also keeps track of the balance of each externally owned account so that frequent traders can save gas by holding their balance in the `Vault`. Another feature arising from this design is the ability to issue flash loans as large as the total value locked in the `Vault`.

The vault only supports classic ERC20 tokens, and specifically does not support tokens with more than 18 decimals, deflationary, rebased or fee-charging tokens.

Relayers

Accounts can grant relayers the permission to deposit, withdraw and transfer funds in and out the `Vault` on their behalf. Relayers will also be able to join and exit `Pools` and make swaps for the underlying accounts.

Pools

`Pools` can create their own trading algorithms meaning that new AMMs designs can be plugged into the `Vault` to give it more capabilities without the need to redeploy nor migrate funds from the `Vault`. The `Vault` keeps track of the `Pool` balance with different data structures according to the specialization of the `Pool`. There are three kinds of pool specializations: `GeneralPools`, `MinimalSwapInfoPools` and `TwoTokenPools` depending on the amount of information being passed in the swap.

Asset Managers

`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.

Roles and their security assumptions

The `Authorizer` contract sets up an admin address and then this value will be used by the `Vault` to create a new `Authorization` instance with `Authorizer` contract address as the input value for its constructor. The admin set up by the `Authorizer` will have the ability to grant or revoke roles, which will impact in the `Authorization` contract's context when this contract queries if the caller has a certain role. The administration address can give permissions to any address to:

- Change the `Authorizer` contract address.
- Set swap, withdraw and flashloan protocol fees

- Withdraw collected fees
- Give relayers the roles so that they can:
    - Deposit to accounts balance
    - Withdraw from internal balance
    - Transfer internal balance to other accounts
    - Join and exit pools
    - Swaps - Batch swap given in and batch swap given out.

Once that the admin grants relayers with the roles, users still need to allow those specific relayers allowing them to opt-out of relayers if they prefer it that way.

Scope

We audited commit `1cb36eb56a6b7dbd70bfa3dc16b53357b43b9d5a` of the Balancer protocol contracts repository. In scope are the smart contracts in the `contracts` directory. However, the `test` directory was deemed as out of scope.

Here we present our findings

Critical severity

None.

High severity

Components of `StableMath` formulas miscalculated

In the `_inGivenOut`, `_outGivenIn` and `_calculateDueTokenProtocolSwapFee` functions of the `StableMath` contract the `n` variable is used to store the number of tokens.

In L108, L154, and L251 the variable `nn` is intended to accumulate the value "n^n", but instead accumulates the value "n^2" due to a bug in its implementation where `nn = totalCoins.mul(totalCoins)` is used instead of `nn = nn.mul(totalCoins)`.

These bugs in implementation of formulas underpinning the `StablePool` are likely to cause numerous accounting errors in the protocol such as mispricing of assets in swaps.

Consider carefully reviewing and reworking the functions in `StableMath` to either exactly implement the specification or approximate it within some acceptable and well-documented margin.

Medium severity

`joinPool` called by non-trivial relayer can revert

The `joinPool` function of the `PoolRegistry` contract can revert when it tries to receive tokens from the caller `msg.sender`. This revert would occur in the case that the `msg.sender` is a relayer authorized by a user whom is not itself, and that does not own the quantity `amountIn` of some token in `tokens`.

This can eliminate the capability of a relayer to act on behalf of a sender in joining a liquidity pool.

Consider changing the parameter from `msg.sender` to `sender` in L269.

Note: This issue was also discovered independently by the Balancer team while we were conducting the audit.

Low severity

Deregistering wrong tokens won't revert

The `PoolRegistry` contract presents the `deregisterTokens` function for allowing pools to deregister tokens that will no longer be used by the pool. Depending on the pool specialization, this function may call either `_deregisterMinimalSwapInfoPoolTokens` or `_deregisterGeneralPoolTokens`.

The first of these functions will require that the token is in the set, while the second one will allow the execution to complete without requiring that the token is in the map.

Considering being consistent on the behavior of the protocol when deregistering tokens across the different types of pools.

`BalancerPoolToken` emits disorderly `Transfer` events

Both the `_mintPoolTokens` and `_burnPoolTokens` functions of the `BalancerPoolToken` contract emit disorderly `Transfer` events.

Calling the `_mintPoolTokens` function emits two `Transfer` events. The first being the `Transfer` event called by the `_move` subroutine and the second on line 139. The first event logs a transfer from the contract to the `recipient`, and the second event logs a transfer from the zero-address to the recipient. This is counterintuitive to the common model of the minting of a token as first a transfer from the zero-address to the contract, and then a transfer from the contract to the recipient

Similarly, calling the `_burnPoolTokens` function emits two `Transfer` events. The first being the `Transfer` event called by the `_move` subroutine and the second on line 151. This first event logs a transfer from the `sender` to the contract, and the second event logs a transfer from the `sender` to the zero-address. Again, this is counterintuitive to the common model of the burning of a token as first a transfer from the `sender` to the contract, and then a transfer from the contract to the zero-address.

The ecosystem that develops around core protocols such as Balancer includes applications at various levels of the stack. Many such applications rely on logs produced by events to trigger actions or locally account for transitions of the system. Disorderly events can impact the efficacy of these applications to interact with Balancer.

Consider refactoring the `_mintPoolTokens` and `_burnPoolTokens` functions to either each emit one transfer from the "source to destination", or each emit two transfers but properly modeling the transfer from "source to intermediary to destination".

Lack of input validation

The `registerTokens` function of the `PoolRegister` contract can fail in line L207 if the `assetManagers` array has length strictly less than that of the `tokens` array. This failure can unexpectedly stop the execution, reverting without any explicit reason.

Following the "fail early and loudly" principle, consider including specific require statements early in the function to validate that the inputs are of the correct length.

Using a one-step transfer of the `_authorizer` role

In the `Authorization` contract, the `changeAuthorizer` function sets a new `_authorizer` to the provided address.

If an incorrect address is set as the `_authorizer`, the `changeAuthorizer` function will not be able to be called again and the logic of the `Authorization` contract modified by `authenticate` will be inaccessible, causing among other consequences, the impossibility to withdraw the fees from the protocol.

When doing such operations, a recommended pattern is to use a two-steps design where the transfer is first initiated and then accepted by the corresponding recipient. In this way, an incorrect value can be overwritten by a new transaction.

Consider using the recommended pattern to avoid consequences of changing the `_authorizer` variables to an incorrect value.

`roleId` may not be unique

The value of the `roleId` of the `_authenticateCaller` function of the `Authorization` contract may not be unique for the functions that it services. In the audited codebase, the functions that `_authenticateCaller` service are `changeAuthorizer`, `setProtocolFees`, and `withdrawCollectedFees` by way of the `authenticate` modifier. The `roleId` is constructed using the `msg.sig` of the serviced functions. Recall that the `msg.sig` consists of the first four bytes of the function signature hash.

As such, a collision can easily occur between the `msg.sig`'s of any two serviced functions. If such a `msg.sig` collision occurs for a pair of serviced functions, granting the role for an account to access one function will necessarily grant the role for the same account to access the other function.

While in the case of this version of Balancer, there is not in fact a collision between the `msg.sig`'s of the serviced functions `changeAuthorizer`, `setProtocolFees`, and `withdrawCollectedFees`, there are not any checks in the test suite for this

vulnerability. Having this check would guard against collisions for functions added or modified before release.

Consider adding tests to the test suite to check for collisions of the `msg.sig`'s of functions having the `authenticate` modifier.

Notes & Additional Information

Additional client-reported issues

During our audit, the Balancer team independently discovered that the `flashLoan` function of the `FlashLoanProvider` contract does not validate that the `tokens` array contains only unique addresses.

This could be exploited to pay less fees than intended and create accounting errors on the `Vault` contract.

Immutable amplification coefficient

The `_amp` variable of the `StablePool` contract is defined as `immutable`.

As such, this behavior would make the usage of this type of pool dangerous for stablecoins, specially for those that tend to depeg easily from its value as we have seen for the most battle-tested implementation of this algorithm in the Curve protocol, where the value of the amplification coefficient is constantly modified by the governance of the protocol.

Even though the Balancer team was already aware of this issue, we decided to include it in the report for future reference on developers that want to create their own version of this algorithm to be aware that if this value is not constantly tuned it can create opportunities for arbitrageurs and in detriment of the liquidity providers of the pool.

Consider making the `_amp` variable non immutable to allow the pool to be tuned on the specific values this variable needs depending on the pegging and depegging of a specific asset to its underlying value.

Incorrect comment documentation on `StableMath.sol`

In the comment located in L32 of the `StableMath` contract one of the factors is defined as `A * n^n` and in the comment located in L34 of the `StableMath` contract one of the factors is defined as `( A * n^n â^’ 1)`.

However, in L58 the first factor stated above is defined as the amplificator times the number of tokens, and thus appears to be lacking `n - 1` factors of `n`.

Similarly, in L59 the second factor stated above is defined as the amplificator times the number of tokens minus one.

This was confirmed by the Balancer team to be a documentation issue which root cause is on the Curve documentation, as the `_A` value is not the amplification coefficient multiplied by "n * (n - 1)" but rather the amplification coefficient multiplied by "$n^{(n - 1)}$".

Consider modifying the documentation to match the current state of the codebase.

Naming issues hinder code understanding and readability

To favor explicitness and readability, many parts of the contracts may benefit from better naming. Our suggestions are:

- Changing the `tokenAmount` variable to `numberOfRegisteredTokens`
- Changing the `_addSwapFee` function to `_includeSwapFee`, since addition operation isn't explicitly performed.
- Changing the `_subtractSwapFee` function to `_removeSwapFee`, to complement `_includeSwapFee` above.

Pools extending `BaseGeneralPool` can lock

If a pool extending the `BaseGeneralPool` contract deregisters a token using the vault's `deregisterTokens` function it can be locked in the sense that users will not be able to join, exit, or swap with the pool.

Deregistering a token by calling `deregisterTokens` will remove the token from its representation in the token's vault. But this deregistering does not remove the token from its representation in the `BasePool` that `BaseGeneralPool` inherits. So the `BasePool` also maintains the `internal` `immutable` `_totalTokens` quantity of tokens that the pool was initialized with.

So for all joining, exiting, and swaps their corresponding function calls, (`joinPool`, `exitPool`, `batchSwapGivenIn` or `batchSwapGivenOut`), will fail within their respective calls to `_upscaleArray`. This is because the bound for the `for` loop in `_upscaleArray` is `_totalTokens` which is now strictly greater than the `amount` array which reflects the tokens' representation in the vault.

Even though the current codebase does not make use of the `deregisterTokens` function, we decided to include this issue for developers building on top of the `BaseGeneralPool` contract.

Consider thoroughly documenting that a pool extending `BaseGeneralPool` should not have the ability to call `deregisterTokens`.

TODOs in the code

There are "TODO" comments in the code base that should be tracked in the project's issues backlog. Examples of this can be found in:

- L82 and L87 of the `Math.sol` file.
- L187 of the `WeightedMath.sol` file.
- L33 of the `WeightedPool.sol` file.

During development, having well described "TODO" comments will make the process of tracking and solving them easier. Without that information, these comments might tend to rot and important information for the security of the system might be forgotten by the time it is released to production.
These TODO comments should at least have a brief description of the task pending to do, and a link to the corresponding issue in the project repository. Consider updating the TODO comments to add this information. For completeness and traceability, a signature and a timestamp can be added. For example: `// TODO:`

`point this at an interface instead. // https://github.com/balancer-labs/balancer-core/issues/XXXX // --<username> - 20201129`

Typographical errors

- L80, L126, and L222 of `StableMath.sol` the comments documentating the Newton-Raphson approximation incorrectly states the coefficient for the first degree term to be `S - D / (A*n^n) - 1` where it should be `D / (A*n^n) + S - D`.
- L183 and L188 of `StableMath.sol`: `amountsOut` should be `amountsIn`.
- L160 of IVault.sol: `IGeneeralPool` should be `IGeneralPool`
- L585 of `IVault.sol`: `tointernalBalance` should be `toInternalBalance`
- L600 of `IVault.sol`: one of the `batchSwapGivenOut` should be `batchSwapGivenIn`
- L550 of `Swaps.sol`: `nnow` should be `now`
- L212 of `BalanceAllocation.sol`: `Becase` should be `Because`
- L124 and L425 of `EnumerableMap.sol`: `whithin` should be `within`
- L34 of `FixedPoint.sol`: `addition` should be `subtraction`
- L32 of `BasePool.sol`: `manges` should be `manages`
- L266 of `StableMath.sol`: `calcuates` should be `calculates`
- L467 and 517 of `IVault.sol`: `* @dev Performs a series of swaps with one or multiple Pools. In individual each swap, the amount of tokens sent to ...` should delete `individual`.
- L42 of `Swaps.sol`: `// Despite the external API having two separate functions for given in and given out, internally their are handled` "their" should be "they".
- L194 of `BasePool.sol`: `MINUMUM_BPT` should be `MINIMUM_BPT`

Unnecessary imports

In the `BasePoolFactory.sol` file, consider removing the import statements for `Address.sol`, `EnumerableSet.sol`, and `IBasePool.sol`, as they are never used in the `BasePoolFactory` contract.

Unnecessary `require` statement

The `_burnPoolTokens` function of the `BalancerPoolToken` contract has an unnecessary `require` statement.

As a result of the call to the internal `_move` function, the `BalancerPoolToken` contract will already have the necessary balance to burn.

Consider removing this unnecessary `require` statement to favor simplicity.

Unnecessary type cast

In L279 of `EnumerableMap.sol` the `key` parameter is being casted to the `address` type and then recasted to the `uint256` type.

Since the `key` parameter is already an `uint256`, the casts to and from `address` are unnecessary.

To simplify the codebase, consider omitting unnecessary casts.