Balancer Contracts Audit

AUGUST 9, 2020 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

Balancer is an automated portfolio manager. It allows anyone to create Balancer pools, each of which implements an automated market maker (AMM) that is a generalization of the constant-product AMM popularized by Uniswap. Each pool can contain a collection of up to 8 ERC20 tokens, value-balanced according to user-defined weights. These pools can be used, for example, to create on-chain index funds without the use of price oracles. You can learn more about Balancer by reading their whitepaper. At the Balancer team's request, we've audit their code and now publish our results.

Scope

We audited commit `f4ed5d65362a8d6cec21662fb6eae233b0babc1f` of the `balancer-core` repo. We audited all files in the `/contracts/` directory with the exception of the `Migration.sol` file and any files in the `/tests/` directory.

Trust Model

BFactory

New Balancer pools can be created by anyone via the `BFactory` contract. Anytime liquidity is removed from a pool, an `EXIT_FEE` is charged and sent to the `BFactory` contract (at the time of writing the `EXIT_FEE` is `0`, and so no exit fee is extracted when liquidity is removed from a pool). The `BFactory` contract has a privileged `_blabs` role that is capable of withdrawing any ERC20-compliant tokens held by the factory contract. This is how exit fees will be collected if/when the `EXIT_FEE` is changed. Other than the ability to withdraw fees from the `BFactory` contract, the `_blabs` role has no other power.

Neither the `BFactory` contract nor the `BPool` contracts that it instantiates are upgradable.

BPool Controllers

Each `BPool` contract has a `_controller` role which defaults to the address that initiated the creation of the pool.

Before a `BPool` is `_finalized`, the `_controller` is capable of trivially stealing most funds from the pool. This can be accomplished, for example, by binding a new token (which they fully control) to the pool, minting a large number of those tokens for themselves, then draining the remaining tokens from the pool by swapping them out for the newly-minted tokens. So for non-finalized pools, users must fully trust the `_controller` of the pool.

Additionally, even a `_finalized` pool can be manipulated (e.g.: DoS'd, drained of its value, etc) if any of its bound tokens have contracts that can be controlled by a third party. For example, a finalized pool that has bound the USDC token could be DoSed or drained of its value if Coinbase (who controls and can arbitrarily update the USDC contract) were malicious. For example, it could inflate USDC and use the new tokens to extract most of the remaining tokens from the pool, or it could put the `BPool` address on its "blacklist", which would prevent any USDC from being transferred out of the pool.

So for both finalized and non-finalized pools, the user must fully trust the contracts of the tokens that have been bound to the pool.

Choice of tokens

As mentioned in the docs:

> Bronze does not support ERC20 tokens that do not return bools for `transfer` and `transferFrom`.

This is because the `_pullUnderlying` and `_pushUnderlying` functions expect the underlying ERC20 tokens to return `true` on successful calls to their `transfer` and `transferFrom` functions. This makes sense because this behavior is required by the ERC20 specification.

It is important to note, however, that there exist tokens that are thought of as "ERC20 tokens" but do not actually follow the specification with regards to returning a `bool` on a successful call to `transfer` or `transferFrom`, and that such non-compliant behavior could result in those tokens being stuck in a `BPool` contract.

For example, the popular BNB token from Binance returns `true` on a successful `transferFrom`, but does not return `true` on a successful `transfer`. This means that BNB could be transferred into a `BPool` contract (because the `_pullUnderlying` function would succeed) but could not be transferred out of a `BPool` contract (because the `_pushUnderlying` function would revert on line 712).

This issue does not affect fully-compliant ERC20 tokens, and so is not technically a break from the specification. However, it may, in practice, result in loss of funds. The developers are aware of this issue and it is mentioned both in the "limitations" section of the docs and in their FAQ.

Furthermore, some popular tokens can extract a fee from the sender and/or the recipient on calls to `transfer` and `transferFrom`. For example, USDT is capable of extracting a fee from the recipient on calls to `transfer` and `transferFrom` (though no fee is being extracted at the time of writing). This non-standard behavior can result in serious accounting errors in the `BPool` contract that may result in loss of funds.

In particular, for any given token bound to the pool, the `BPool` contract tracks its token balance in a `Record` struct in the `_records` mapping. The `balance` of this `Record` gets updated (for example, here) before each call to the `_pullUnderlying` and `_pushUnderlying` functions. However, the code that updates the `balance` assumes that the amount sent and received is exactly equal to the `amount` parameter used when calling the `transfer` or `transferFrom` functions — that is, it assumes standard balance-updating behavior of the underlying token. So for non-standard tokens of this type, the `BPool` contract may actually hold more or fewer tokens than its corresponding `Record` would suggest. This error in accounting may be used to steal value from the pool.

To add support for these two types of non-compliant and non-standard tokens in future versions of Balancer, consider using OpenZeppelin's `SafeERC20` library to perform the underlying token transfers. This library handles the edge cases where otherwise-compliant tokens don't return a `bool` on a call to `transfer` or `transferFrom`. Also consider using the "before and after balance check" pattern (see Compound's `doTransferIn` function for an example) to handle the case where tokens do non-standard balance updating during a `transfer` or `transferFrom`. This pattern measures the amount of a given token that has been added or removed from the pool, as opposed to relying on the `amount` parameter of the `transfer` and `transferFrom` functions of the underlying tokens.

Alternatively, consider allowing users to make arbitrary deposits & withdraws from the `BPool` contract, then at the end of the function call, check if the token balances of the `BPool` contract are acceptable, and revert if not (see UniswapV2's `swap` function for an example). This puts the burden of proper accounting on the user (or intermediary contract), while the `BPool` contract itself needs only verify that its own token balances are acceptable — without having to consider any possible non-standard balance-updating behavior of the underlying tokens.

General health

Overall we found the Balancer contract code to be clear and well-written. Its component parts can be easily analyzed and tested in isolation. The whitepaper and documentation were helpful and the repo includes extensive tests.

Findings and Recommendations

Here we report our findings and recommendations.

Critical severity

None.

High severity

None.

Medium severity

None.

Low severity

Rounding errors are not always in favor of the pool

Fixed-point arithmetic introduces small errors in the computations used throughout the contracts. The `BNum` library makes use of fixed-point algorithms that minimize these errors (by rounding to the nearest smallest unit). However, these errors still exist and, importantly, are not always in favor of the pool. This means it may be possible for users to illegitimately remove dust amounts of from the pool. This can happen both when adding and removing liquidity and when performing swaps. Under extreme conditions it may be profitable to exploit these errors.

As an example, consider the `joinPool` function. Suppose the `poolTotal` is `100e18` and one of the bound tokens, `t`, has a record such that `_records[t].balance = 20000005`, which represents a token balance of `200.00005` for a token with 5 decimals. If the user supplies a `poolAmountOut` parameter value of `17857142860000000000` (approximately 17.86e18), then the `ratio` variable will be `178571428600000000`. If there were no rounding errors, we would expect that the user would have to transfer in `35.71429464857143` units of the token. However, due to the rounding error introduced by `bmul`, the user actually transfers in `35.71429` tokens, which is `0.00000464857143` fewer tokens than expected. In other words, the user bought shares of the pool slightly more cheaply than intended. The error introduced by the fixed-point arithmetic, though small, was not in favor of the pool in this case.

Similar situations can arise when exiting the pool, and when interacting with the swapping functions. Computing the maximum error bounds for the swapping functions is more difficult (we did not explicitly compute them during this audit), but the same principles may apply, and the underlying issue is that the errors may not always favor the pool.

Under most normal conditions, these errors are negligible and not profitable to exploit. The cost of gas alone would typically make exploitation cost-prohibitive, and the error must be great enough (and in favor of the attacker) to outweigh any swap fees or exit fees incurred.

In some very extreme cases, however, exploiting these errors may be profitable. For example, if an attacker can illegitimately remove an extra `0.000004` tokens from the pool, and those tokens are worth $1M each, then the attacker could gross $4 per transaction. If it cost less in gas to perform the attack (including the gas cost of flash-borrowing the required capital if necessary) then you could expect attackers to create bots to exploit the rounding errors until the pool was emptied.

All else being equal, these errors become more severe when the tokens in question have fewer decimals, and when the value of the whole units of the token are greater.

Consider refactoring the code to ensure that all arithmetic errors related to moving tokens are in favor of the pool.

For example, in the `joinPool` function, consider adding `1` to the output of `bdiv` on line 358 to counteract any rounding-down that may have ocurred during the execution of `bdiv`, and consider adding `1` to the output of `bmul` on line 381 to counteract any rounding-down that may have ocurred during the execution of `bmul`. This would ensure that all calls to the `joinPool` function have arithmetic errors that are in favor of the pool. While this may slightly increase the absolute size of the errors, it enforces that the *direction* of the errors favors the security of the pool.

With all arithmetic errors in favor of the pool, there would be no need to worry about attackers splitting/batching transactions in an attempt to exploit arithmetic errors in their favor, even under extreme conditions.

Notes & Additional Information

Missing content and broken links in the documents

There are a few broken links and some missing content in the official docs. For example under the API section, the `getCurrentTokens()->[Ts]` and `getFinalTokens->[Ts]` links both point to the `getNumTokens` section of the docs, and the content for those functions is missing from the page.

Consider double checking the docs for missing content and broken links.

Possible gas improvements

In their whitepaper, Balancer mentions that the Silver release will focus, in part, on gas improvements. Looking for gas improvement opportunities was not in-scope for this audit. However, we noticed a couple of these opportunities in passing and mention them here for convenience. This is not a complete or thorough list of gas improvement opportunities.

- There are instances where memoization could improve gas performance. For example, the `increaseApproval` function could memoize the `_allowance[msg.sender][dst]` variable to improve gas performance. E.g.:

```
    function increaseApproval(address dst, uint amt) external returns (bool) {
        uint oldAllowance = _allowance[msg.sender][dst];
        _allowance[msg.sender][dst] = badd(oldAllowance, amt);
        emit Approval(msg.sender, dst, oldAllowance);
        return true;
    }
```

- The `require` statements on lines 51 and 58 of `BToken.sol` are not necessary because the `bsub` on the next lines will revert if there is an underflow. Removing these `require` statement improves gas performance for non-reverting calls to `_burn` and `_move`, but reduces gas performance for calls that would underflow. Since the former is likely to be much more common than the latter, this could be a net positive for gas performance.

- In some places, pool shares are minted to the `BPool` contract and then pushed out to a user. It may be more efficient to mint the shares directly into the user's account. This would require refactoring the `mint` function.

Incorrect code comments in BMath.sol

The functions in the `BMath` contract are correct implementations of the functions described in the whitepaper and the code comments for the various functions in `BMath.sol` are correct, with the following exceptions.

- The `tO` variable in the comments of the `calcPoolInGivenSingleOut` function should be a `wO`.

- The `wO` variable (which represents the `tokenWeightOut`) is not defined in the comments of the `calcSingleOutGivenPoolIn` function.

- The `wI` variable is defined in the comments of the `calcSingleOutGivenPoolIn` function even though it is not used in the function.

- There is a `b0` in the comments of the `calcSingleOutGivenPoolIn` that should be a `bO` (that is, it should be the letter "oh" instead of the number zero).

These errors affect only the code comments and are not present in the code itself. Consider correcting these errors to prevent any confusion they may cause readers of the code.

Additionally, the code describing the `calcSingleInGivenPoolOut` function does not match the code itself. The numerator is stored in the `tokenAmountInAfterFee` variable and the denominator is stored in the `zar` variable. So for the code to match the description in the comments and the docs, the return value should be `bdiv(tokenAmountInAfterFee, zar)`. But the actual return value is `bdiv(tokenAmountInAfterFee, bsub(BONE, zar))`. Some internal code comments explain the discrepancy and suggest that the actual behavior of the code is the intended behavior.

Consider adjusting both the code comments and the docs for the `calcSingleInGivenPoolOut` function to make them consistent with the actual behavior of the code for this function.

Lack of code comments in BNum library

While the intended behaviors of the functions in `BNum.sol` are clear, the contract could benefit from NatSpec comments to all of the functions, specifically identifying which `uint` parameters are meant to be interpreted as integers, and which are meant to be interpreted as integer representations of fixed-precision numbers.

Use of named return values

There is an inconsistent use of named return variables. For example, every function in `BMath.sol` has named return variables. None of the functions in `BNum.sol` have named return variables. Some functions in `BPool.sol` have named return variable, while others don't.

Consider removing named return variables from all functions wherever possible. This can help reduce the chances of introducing regressions when refactoring the code in the future.

Possible improvements to the BNum library

The `bpowApprox` function computes `base^exp` using a binomial approximation (specifically, using a Taylor Series approximation about the point zero). This approximation technique is accurate only if the absolute value of `x` (on line 134) is less than `BONE`. An equivalent condition is `0 < base < 2 * BONE`.

This `bpowApprox` function is currently called only via the `bpow` function, which ensures that `0 < base < 2 * BONE`, so currently, all calls to the `bpowApprox` function have their inputs checked to ensure the function is used only with valid inputs.

However, there may be some room for improvement here.

Consider removing the two `require` statements from the `bpow` function (because they aren't strictly needed unless `exp` is not a whole number) and instead putting a `require(x < BONE)` statement after line 134 of `bpowApprox`. Additionally, if the `bpowApprox` function is intended to be used only for values of `exp` less than `BONE`, consider adding a `require(exp < BONE)` statement to the beginning of the `bpowApprox` function.

This would allow the `bpowApprox` function to be used safely, all on its own, independent of how it used by the `bpow` function. That is, it would ensure that the necessary limitations for the binomial approximation are always enforced by the `bpowApprox` function itself, instead of relying on developers to remember to perform those checks on their own before calling the `bpowApprox` function. It would also remove unnecessary limitations on the `bpow` function when using it with whole-number exponents.

In general, this would make the `BNum` library more flexible and less prone to error during future code refactors.

Use of aliases

To favor explicitness, consider declaring all instances of unsigned integers using `uint256` instead of the alias `uint`.

Security Audits

RELATED POSTS

SECURITY AUDITS

Aave Protocol Audit Summary

The Aave team asked us to review and audit a pre-production version of their protocol.

READ MORE

by OpenZeppelin Security

SECURITY AUDITS

Opyn Contracts Audit

The Opyn team asked us to review and audit the system. We looked at the code and now publish our...

READ MORE

by OpenZeppelin Security

SECURITY AUDITS

Augur Core Audit

The Augur team asked us to review and audit their Augur Core project. We looked at the code and now...

READ MORE

by OpenZeppelin Security