OpenZeppelin | security

Bancor Protocol v3 Audit

Bancor

June 2nd, 2022

This security assessment was prepared by **OpenZeppelin**, protecting the open economy.

Table of Contents

Summary

The Bancor team asked us to review the smart contracts for version 3 of their protocol, which is described by BIP15. This includes the single-sided pools, withdrawal algorithm, and staking rewards mechanics.

Type	DeFi, Decentralized Exchange
Timeline	Audit Start: 2022-03-23 Audit End: 2022-05-03 Fixes Finalized: 2022-06-02
Languages	Solidity

Total Issues	18 (10 resolved)
Critical Severity Issues	0
High Severity Issues	0
Medium Severity Issues	2 (2 resolved)
Low Severity Issues	5 (0 resolved)
Notes & Additional Information	11 (8 resolved)

Scope

We audited commit a58f36b of the bancorprotocol/contracts-v3 repository.

The following contracts constitute the scope of the audit:

```
contracts
├── bancor-portal
│   ├── BancorPortal.sol
│   └── interfaces
│       └── IBancorPortal.sol
├── network
│   ├── BancorNetwork.sol
│   ├── BancorNetworkInfo.sol
│   ├── BancorV1Migration.sol
│   ├── NetworkSettings.sol
│   ├── PendingWithdrawals.sol
│   └── interfaces
│       ├── IBancorNetwork.sol
│       ├── IBancorNetworkInfo.sol
│       ├── INetworkSettings.sol
│       └── IPendingWithdrawals.sol
├── pools
│   ├── BNTPool.sol
│   ├── PoolCollection.sol
│   ├── PoolCollectionWithdrawal.sol
│   ├── PoolMigrator.sol
│   ├── PoolToken.sol
│   ├── PoolTokenFactory.sol
│   └── interfaces
│       ├── IBNTPool.sol
│       ├── IPoolCollection.sol
│       ├── IPoolMigrator.sol
│       ├── IPoolToken.sol
│       └── IPoolTokenFactory.sol
├── staking-rewards
│   ├── AutoCompoundingStakingRewards.sol
│   ├── StakingRewardsMath.sol
│   ├── StandardStakingRewards.sol
│   └── interfaces
│       ├── IAutoCompoundingStakingRewards.sol
│       └── IStandardStakingRewards.sol
├── token
│   ├── ERC20Burnable.sol
│   ├── SafeERC20Ex.sol
│   ├── Token.sol
│   ├── TokenLibrary.sol
│   └── interfaces
│       └── IERC20Burnable.sol
│
```

```
├── utility
│   ├── BlockNumber.sol
│   ├── Constants.sol
│   ├── FractionLibrary.sol
│   ├── MathEx.sol
│   ├── Owned.sol
│   ├── Time.sol
│   ├── TransparentUpgradeableProxyImmutable.sol
│   ├── Types.sol
│   ├── Upgradeable.sol
│   ├── Utils.sol
│   └── interfaces
│       ├── IOwned.sol
│       ├── IUpgradeable.sol
│       └── IVersioned.sol
└── vaults
    ├── ExternalProtectionVault.sol
    ├── ExternalRewardsVault.sol
    ├── MasterVault.sol
    ├── Vault.sol
    └── interfaces
        ├── IExternalProtectionVault.sol
        ├── IExternalRewardsVault.sol
        ├── IMasterVault.sol
        └── IVault.sol
```

The Bancor protocol will also implement a vortex module, which will be used to help manage the Bancor network token value. Note that this vortex module has not yet been written for Bancor v3 as of the audit commit and is out of scope for this audit.

System overview

Bancor v3 is a decentralized exchange that allows users to provide liquidity while benefiting from protection against price divergence of tokens in liquidity pools (also known as impermanent loss). The insurance guaranteed to liquidity providers is achieved by balancing fees collected on token swaps and withdrawals against the price divergence loss that otherwise would be realized upon withdrawal.

The Bancor Network Token (`BNT`) facilitates loss protection to liquidity providers for all pools. Every liquidity pool is single-sided, where a base token (hereafter referred to as `TKN`) is paired against the protocol-controlled `BNT`. Liquidity pools follow a standard constant product curve (adjusted for protocol fees). Bancor users can swap between `BNT` and any allowed `TKN`, via a 1-hop trade. Users can also swap between any two allowed `TKN`s via a 2-hop trade, which is functionally equivalent to two 1-hop trades, from `TKN1` <-> `BNT` <-> `TKN2`. In addition to swaps, the Bancor exchange also offers users atomic flash loans for a fee.

Base token liquidity providers receive `bnTKN` tokens which represent their staked balance in the `TKN` liquidity pool. When a liquidity provider decides to withdraw their `TKN` (after a 7-day cooldown period), they will receive their entire staked principal, plus accrued fees and rewards, in `TKN` if possible. When it is not possible to reimburse a liquidity provider entirely in `TKN` (e.g., due to technical constraints), they will instead receive a portion of their `TKN` due and `BNT` equivalent to the value of the remaining `TKN` they did not receive.

In addition to staking `TKN`, users can also stake `BNT` (in exchange for `bnBNT` pool tokens) to share in the trading fees generated by any of the `TKN` pools. All `BNT` is treated equally in the protocol, used to provide liquidity to all of the `TKN` pools; it is not possible to stake `BNT` and only participate in a specific `TKN` pool. Therefore, `BNT` stakers participate in all `TKN` pools. Withdrawing a `BNT` stake is simply done by exchanging `bnBNT` for the underlying `BNT` tokens.

Fees

Various fees are imposed on different functions to incentivize liquidity, mitigate security risks (e.g., by making certain arbitrage opportunities infeasible), and accrue value to `BNT` token owners.

A trading fee taken from the target asset is imposed on all swaps. For swaps between two different base tokens, the trading fee is imposed on each leg of the swap. Trading fees are applied by adjusting the staking ledger balance to increase the value of `bnTKN` by the trading fee amount. A portion of the trading fees are withheld from the staking ledger balance and will instead be used to increase the vortex ledger balance. A vortex module, yet to be implemented, also uses the trading fees. The vortex balance will subsequently be used to buy back `vBNT` (the `BNT` governance token) for burning to deflate the value of `BNT` for the benefit of owners. In addition, flash loans will incur a fee denominated in the token borrowed, with the proceeds directly increasing `TKN` vault balances. Upon withdrawal, liquidity providers will also incur a fee. The withdrawal fee is removed from the withdrawal proceeds and is an integral part of the described withdrawal algorithm below.

Withdrawal algorithm

`BNT` liquidity withdrawals are straightforward to process. Liquidity providers can simply sell their `bnBNT` pool tokens for an equivalent amount of `BNT` tokens.

Withdrawals of `TKN` are much more complex, following different logic depending on the financial state of the protocol. When the protocol has a deficit amount of `TKN` (i.e., `TKN` has appreciated relative to `BNT`), a withdrawal will create an arbitrage opportunity to attempt to recapture `TKN` from external markets by repricing `BNT` against `TKN` in the pool. When the protocol has a surplus of `TKN` (i.e., `TKN` has depreciated relative to `BNT`), a withdrawal will create an arbitrage opportunity to attempt to eject `TKN` into external markets by a different repricing of `BNT` against `TKN` in the pool. If the arbitrage opportunity necessitated by a withdrawal exceeds the withdrawal fee, the liquidity provider will receive a portion of their `TKN` due and `BNT` equivalent to the value of the remaining `TKN` they did not receive, to prevent abuse of the algorithmic withdrawal system. In all instances, the withdrawal algorithm is designed to:

- maintain the relative surplus or deficit, preceding and following the withdrawal
- maintain liquidity, preceding and following the withdrawal
- pay liquidity providers in their base token

While the withdrawal contracts are well commented, they are insufficient to understand the withdrawal algorithm details well. It is necessary to review the accompanying whitepaper. We encourage the Bancor team to continue to build out documentation and open it to the public.

Bootstrapping pools

Only approved tokens can be used to create liquidity pools within the Bancor network. Once the Bancor DAO approves a new `TKN` for trading, the protocol can begin accepting liquidity stakes. Once enough `TKN` has been staked, trading pools can be initialized. Additional staking actions will increase the pool liquidity up to the predefined funding limit.

Privileged roles

There are numerous roles used in the protocol, generally set to specific contract addresses within constructors. Below is a list of roles currently defined in the system, associated with their capabilities:

- `ROLE_ASSET_MANAGER` - create staking rewards programs, withdraw funds from external protection vaults
- `ROLE_BNT_MANAGER` - mint and withdraw BNT
- `ROLE_BNT_POOL_TOKEN_MANAGER` - withdraw bnBNT tokens
- `ROLE_EMERGENCY_STOPPER` - pause and resume the Bancor network

- `ROLE_FUNDING_MANAGER` - requests and renounces BNT funding for approved `TKN` pools
- `ROLE_MIGRATION_MANAGER` - migrate liquidity between pool versions and types
- `ROLE_NETWORK_FEE_MANAGER` - withdraw network fees
- `ROLE_VAULT_MANAGER` - burn BNT

There are also several critical variables that can be set, either by a contract admin or owner. For example, there are functions to set (update) the trading fee, vortex fee and rewards, flash loan fee, withdrawal fee, withdrawal lock duration, funding limits and minimums for pool liquidity.

Security model and trust assumptions

The Bancor protocol is controlled by an active DAO.

In addition to updating the system via previously mentioned sensitive variables, the protocol is also arbitrarily upgradeable. The DAO has unrestricted control over the protocol, and we assume that the DAO is acting in the best interest of users and the system.

The economic stability also relies on balancing the protocol revenue generated (e.g., via swaps, flash loans, and withdrawals) against the insurance paid to liquidity providers. It is necessary to ensure that fees are set such that the protocol remains a going concern, despite whatever market conditions may arise. We assume the DAO will be active in monitoring the financial health of the system and updating fees, should this ever be necessary.

The DAO also has control over which base tokens Bancor can handle. Some tokens in the wild may be malicious. We trust that the DAO will not approve base tokens that are malicious or otherwise incompatible with the protocol.

Findings

Here we present our findings.

Medium Severity

M-01 Double counting `networkFee`

The `BancorNetwork._tradeBNT()` function is in charge of calling `_bntPool.onFeesCollected()` on line 1194 to collect the trading fees for swaps with BNT as the target.

However the function `onFeesCollected()` was called with the parameter `feeAmount` as `tradeAmountsAndFee.tradingFeeAmount - tradeAmountsAndFee.networkFeeAmount` in line 1196. The problem is that the `networkFee` was already removed from `tradingFeeAmount` in `PoolCollection._processNetworkFee()` on line 1468 and returned in the `result` struct as `result.tradingFeeAmount`.

Due to the call to `_bntPool.onFeesCollected()`, the protocol has removed double the amount of `networkFee`.

Consider correcting the double-counted network fee amount.

Update: *Fixed in commit `83c07a8db7186f12b1bc42afab3b03958e0790fc`. This issue was also independently found seperate from our audit and fixed before our finding was reported to the client.*

M-02 Rounding in `reducedFraction` causes invalid fractions

The `reducedFraction` function can make valid fractions invalid, in particular, when `fraction.d * max < fraction.n`. As a simple example, calling `reducedFraction` with parameters `fraction = Fraction({ n: 9, d: 2})` and `max = 4` returns `Fraction({ n: 3, d: 0})`.

The `reducedFraction` function is only used in `toFraction112`, where the `max` is specified as `type(uint112).max`. Therefore, in order for `reducedFraction` to return an invalid fraction, `fraction.n / fraction.d` need be greater than `type(uint112).max`, which is a very large order of magnitude.

The `toFraction112` function is used to reduce the `weightedAverage` in the `_calcAverageRate` function, and to reduce the `fundingRate` in the `enableTrading` function.

Since Bancor uses an exponential moving average, `_calcAverageRate` is used to update itself. It is also used to determine whether a liquidity pool has a stable swap rate, which could potentially block withdrawals and trading liquidity updates. Given that the average rate is used to update itself, it's highly unlikely the value would change by an order of magnitude greater than `type(uint112).max` unless something unusual were to happen to one of the tokens (e.g., a very large rebasing).

To `enableTrading` in a pool, a `bntVirtualBalance` and `baseTokenVirtualBalance` are provided to specify the `fundingRate`. In order for the `fundingRate` to initialize the pool with an invalid `averageRate`, the virtual balances provided would need to be different by an order of magnitude at least `type(uint112).max`. This is most likely to happen as a result of user error during input.

Although `reducedFraction` only returns invalid fractions in edge cases, consider mitigating this exposure by using the `isValid` function from the `FractionLibrary`, or modifying the behavior of the `reducedFraction` function such that it does not convert fractions to invalid.

Update: *Fixed in commit* `bc4684180a8b3fb28ad730aa9426a30cf7b63c04`.

Low Severity

L-01 Vault.sol is missing upgradeable slots

The contract `Vault.sol` is an `Upgradeable` contract. However, the `__gap` variable is missing, which reserves storage slots for new variables in upgraded versions.

Consider adding a `__gap` variable to `Vault.sol` to adhere to the upgradeable pattern.

Update: *Not Fixed. The client decided to postpone resolving this issue until the DAO decides to update these contracts.*

L-02 Potential for stuck ETH

While reviewing the codebase, we found that the following contracts have a `receive payable` function to receive ETH but no way to withdraw the entire balance of the contract:

- BancorNetwork.sol
- BancorPortal.sol
- BancorV1Migration.sol
- StandardStakingRewards.sol

Each contract needs the `receive` capability for one or more of its functions to work properly. However, these functions only transfer a specified amount out of the contract. If ETH is accidentally sent to one of these contracts directly, it cannot be withdrawn.

To prevent ETH from getting stuck, consider adding access-controlled functionality that will sweep any remaining balance after an action.

Update: *Not Fixed. Since it is not possible to prevent any ERC20 token from being sent to a contract, the client decided to treat such cases universally by preserving the DAO's ability to introduce the capability to extract any stuck ETH or tokens from the contracts in the future.*

L-03 Lack of validation

We identified the following instances of missing validation in the codebase:

- `transferOwnership` does not check whether the `ownerCandidate` is a `validAddress`.

Consider adding more validation to ensure the protocol will run as intended and reduce the chance of user error.

Update: *Not Fixed. The client stated that this is necessary so that an ownership could be revoked and transferred to 0x0 and is the intended behavior.*

L-04 Missing events

The following operations do not emit events:

- `receive` everywhere, like in the BancorPortal, StandardStakingRewards, BankcorNetwork, BancorV1Migration, and Vault contracts
- `transferOwnership`
- `setTokenSymbolOverride`
- `setTokenDecimalOverride`

Consider emitting events for all state changes and other functionality that would be useful to track off-chain.

Update: *Not Fixed. The client decided not to add these events as they are currently unneeded for backend and integration systems to save gas cost.*

L-05 Missing docstrings

There are a lot of functions in the code base missing or lacking documentation. This lack of clarity hinders a reader's understanding of the code's intention and makes it harder to assess security and correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned, and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Not Fixed. The client currently believes that the code is properly and fully documented. The client found some docstrings redundant (e.g., "@param token the token"), and instead focused on ensuring adequate documentation to understand code behavior.*

Notes & Additional Information

N-01 User deposit amount might be less than the record

When making an ERC20 token deposit with function `_depositToMasterVault()` the system takes the given `amt` and uses `safeTransferFrom()` to move tokens to `_masterVault`, then use the given `amt` as the final deposit to update the pool's `stakedBalance` and mint pool tokens.

However, if the token charges a fee during transfer, it will cause the protocol to receive less than the given `amt`, affecting the records' accuracy.

This is not a concern currently since all tokens must first be whitelisted, and the Bancor team has confirmed they will not be supporting any tokens with transfer fees. However, it is worth noting that an upgradeable token could start charging a fee or tax during transfer at a later stage.

Consider using the delta value of before and after balance as the final `amt` of token received.

Update: *Acknowledged. The Bancor team states:*

> We are fully aware that "rogue" tokens may misbehave in ways that many DeFi protocols will find unexpected (e.g., transfer on fees, upgradable tokens, rebasing, reflections, etc.). Therefore, every token is subjected to explicit due diligence and the whitelisting process by the DAO, and the DAO also reserves the option to disable or remove these tokens in the future.

N-02 block.timestamp may not be reliable

The BancorV3 codebase uses the `block.timestamp` as part of the calculations for staking rewards and time checks for withdrawals. Timestamps can be slightly altered by miners to favor themselves in contracts that have time-dependent logic.

Consider taking into account this issue and warn the users that such scenarios could happen. If the alteration of timestamps cannot affect the protocol in any way, consider documenting the reasoning and writing tests enforcing that these guarantees will be preserved even if the code changes in the future.

Update: *Acknowledged. The Bancor team states:*

> We know that block.timestamp is unreliable in short periods (or even not guaranteed to progress monotonously), but the protocol currently doesn't rely on these assumptions.

N-03 Code path is unreachable

In migratePools, line 495 checks an impossible case given the current code in the `PoolMigrator` contract. `poolMigrator.migratePool` returns the address of the `newPoolCollection` for the supplied pool. In the case where this value is the `address(0)`, the internal call to `_migrateFromV1` will revert when the last line is executed on the `address(0)`.

Consider removing unreachable code paths and redundant checks to improve code legibility.

Update: *Fixed in commit `1431415230bcee2b81d39dfc9c26438ebd613c97`.*

N-04 Inconsistent explicit override for base interface functions

In the `Vault` contract, the function `withdrawFunds` explicitly overrides the base interface function whereas the function `burn` does not explicitly override its base interface function.

For consistency, consider either explicitly overriding all base interface functions or removing all explicit base interface overrides.

Update: *Fixed in commit `3913d4aad5c8992db992c845bf9277bf4947e23b`.*

N-05 Inconsistent naming

There are a few places in the codebase with inconsistent and confusing naming. For example:

- The private `_transferOwnership` function name is confusing, because it is not used by `transferOwnership` and only used by `acceptOwnership`.

- The `migrateSushiSwapV1Position` function emits the `SushiSwapV2PositionMigrated` event.

Consider revising these and other naming decisions to make the codebase easier to understand.

Update: *Fixed in commit* `b07cc845429f83b53b1cb43a51790ef2f78b2a2f`.

N-06 Misleading and confusing comments

We have identified the following comments which may be interpreted as misleading or confusing:

- A comment in `migratePool` states, "it's currently not possible to add two pool collections with the same version or type". However, the logic in `addPoolCollection` validates that a pool collection does not have the same version *and* type. Consider changing the "or" in the `migratePool` comment to "and".
- A comment in the `_migrateFromV1` function states "since the latest pool collection is also v1, currently not additional pre- or post-processing is needed". However, it appears this function can be used to migrate to future pool versions (exceeding version 1), so this comment is confusing as to the capabilities of this function.
- The comments provided for the exponentiation of each binary exponent in the `exp` function are ambiguous and confusing with respect to the order of operations. For example, `e^2^(+3)` actually represents `e^8`, but it might be misinterpreted as `e^6` because of the placement of parenthesis. Consider rearranging the parenthesis, so that it is not ambiguous as to the actual operation, such as `e^(2^+3)`.
- The comment in `completeWithdrawal` when sending poolTokens back to the caller should say that the poolTokens are sent to BancorNetwork contract. Otherwise, it is easy to confuse the "caller" here as the original `msg.sender` (i.e., the liquidity provider).

Consider revising comments to remove ambiguity for others reviewing the codebase.

Update: *Partially Fixed in commit* `6533908fbf3ce3494386a9ebd3be9b32b45fff04`.

N-07 Readability for large numbers

Throughout the codebase there are a few hardcoded large numbers. These numbers can be difficult to read at first glance since there is no delimiter grouping digits. Examples that we have found:

- LAMBDA_N = 142857142857143 => LAMBDA_N = 142_857_142_857_143
- LAMBDA_D = 10000000000000000000000 => LAMBDA_D = 10_000_000_000_000_000_000_000
- PPM_RESOLUTION = 1000000 => PPM_RESOLUTION = 1_000_000
- DEFAULT_TRADING_FEE_PPM = 2000 => DEFAULT_TRADING_FEE_PPM = 2_000
- MAX_RATE_DEVIATION_PPM = 10000 => MAX_RATE_DEVIATION_PPM = 10_000

Consider using underscores (_) as delimiters to improve the readability of hardcoded large numbers.

Update: *Fixed in commit* `c61b1740742561bfef29f3bade845f3e8cdc1b71`.

N-08 Inconsistent use of hardcoded values

In the contract `PoolCollection`, the events `TradingEnabled` and `DepositingEnabled` are emitted at the end of the `createPool` function. `TradingEnabled` uses a hardcoded boolean value for its `newStatus` parameter while `DepositingEnabled` uses the value from the newPool object even though it will always be `true`.

For consistency, consider using the hardcoded values in cases where the value will always be the same.

Update: *Fixed in commit* `a98b9062e6dbf04dc1a86ee7f2941a3e0c88e121`.

N-09 Contracts missing the keyword `abstract`

We found a few instances of contracts that are missing the label `abstract`:

- Utils.sol
- Time.sol
- BlockNumber.sol

Consider making these contracts `abstract` to indicate that they are intended to be used as base contracts.

Update: *Fixed in commit* `71f6a1708de9beb9706c789f906ab40f65b46d20`.

N-10 Typographical errors

The following typographical errors were identified in the codebase:

- "an token" should be "a token". There is are instances of this typo in the `toIERC20` function, and in the `toERC20` function.
- There is an extra closing parenthesis in the comment for the `createdAt` value of the `WithdrawalRequest` struct.

To improve readability, consider correcting typographical errors in the codebase.

Update: *Fixed in commit* `f4bb9b3f06c99b9e0af1f0ece5957459cdccd4da`.

N-11 Gas optimizations

Possible gas cost improvements were found throughout the codebase. In particular:

- The calculations for `currTimeElapsed` and `prevTimeElapsed` in _tokenAmountToDistribute contain unnecessary subtractions of `p.startTime` for a `FLAT_DISTRIBUTION`, since these values are **only** used on line 384 where `prevTimeElapsed` is subtracted from `currTimeElapsed`.
- For loops that have hardcoded conditions, such as `i < 8`, can be turned into while loops where the increment of `i` is performed at the end of the loop inside an `unchecked` block. The following are instances found that can be changed:
  - BancorPortal.sol
  - BancorPortal.sol
  - BancorV1Migration.sol
  - MathEx.sol

To reduce the gas consumption during the execution of the code, consider updating the code to be more performant.

Update: *Not Fixed. The client decided to not implement a fix due to insignificant gas savings.*

Conclusions

2 medium severity issues were found in this audit. Other changes were proposed to follow best practices and reduce the potential attack surface.