Bancor Compounding Rewards Audit

OPENZEPPELIN SECURITY | SEPTEMBER 6, 2022 Security Audits

Bancor Compounding Rewards Audit

August 4, 2022

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

Table of Contents

- Table of Contents
- Summary
- Scope
- Rewards system overview
  - Privileged roles
- Security model and trust assumptions
- Findings
- High Severity
  - reducedFraction does not reduce in the mathematical sense
- Medium Severity
  - weightedAverage calculation might overflow

- Low Severity
  - Unbounded number of programs could result in errors
  - Duplicated code
  - uint32 times can overflow in the future
  - Lack of event emissions after updating state
  - enableProgram is overloaded
- Notes & Additional Information
  - Confusing use of assert statement
  - Unused import
  - Missing docstrings
- Conclusions

Summary

The client asked us to review the smart contracts related to the auto-compounding rewards module, including the mathematical dependencies and how the pool contracts are used by the rewards module.

Type
: DeFi

Timeline
: From 2022-06-06
: To 2022-06-17

Languages
: Solidity

Total Issues
: 12 (2 resolved, 2 partially resolved)

Critical Severity Issues
: 0 (0 resolved)

High Severity Issues
: 1 (1 resolved)

Medium Severity Issues
: 3 (0 resolved)

Low Severity Issues
: 5 (1 partially resolved)

Notes & Additional Information
: 3 (1 resolved, 1 partially resolved)

the bancorprotocol/contracts-v3 repository.

In scope were the following contracts:

```
contracts
├── pools
│       ├── BNTPool.sol
│       |       └── poolTokenAmountToBurn()
│       └── PoolCollection.sol
│               └── poolTokenAmountToBurn()
├── rewards
│       ├── AutoCompoundingRewards.sol
│       ├── RewardsMath.sol
│       └── interfaces
│               └── IAutoCompoundingRewards.sol
└── utility
        ├── Constants.sol
        ├── Fraction.sol
        ├── FractionLibrary.sol
        └── MathEx.sol
```

All contracts not explicitly mentioned above are out of scope for this audit. The `pools` contracts were only audited to the extent that `AutoCompoundingRewards.sol` uses `poolTokenAmountToBurn`.

Rewards system overview

Token projects, including Bancor's `BNT`, can create a rewards schedule by providing liquidity to a pool and then burning the pool tokens it has received at a predetermined rate. Thus, the rewards budget is used as liquidity, and the ownership of the staked tokens is distributed to other stakers over time.

To initialize a rewards program, `TKN` tokens are added to the pool, and `bnTKN` pool tokens are issued as normal (or `bnBNT` for the `BNT` token). Then, the pool tokens are staked into a generic

Interactions with the Vortex contract cause the pool tokens to be burned in accordance with the parameters set by the rewards contract; the Vortex trigger also triggers the rewards distribution. As pool tokens are burned, the token value they represent is effectively distributed to the remaining pool tokens. Thus, no contract interactions are required by stakers; so long as they hold the pool token, they are automatically participating in the rewards program.

The auto compounding system can be used in any instance where the reward token is the same as the staked token; it cannot be used to distribute `BNT` to `TKN`, or to distribute `TKN` to any pool other than itself. For use cases where the reward token is different than the staked token, a standard rewards contract is available. The standard rewards contract is out of scope for this audit.

Privileged roles

The `AutoCompoundingRewards` contract relies on the `ROLE_ADMIN` role and `onlyAdmin` modifier to protect the following functions:

- `setAutoProcessRewardsCount` – Sets the number of programs to auto-process the rewards for each time `autoProcessRewards` is called
- `terminateProgram` – Terminates a rewards program
- `enableProgram` – Changes a rewards program status between enabled and disabled
- `createProgram` – Creates a rewards program for a given pool

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.

Findings

Here we present our findings.

libraries, in particular, we consider how these may be improved upon for safe reuse in potential future iterations or new applications.

High Severity

`reducedFraction` does not *reduce* in the mathematical sense

In the `MathEx` library, the function `reducedFraction` does not reduce the Fraction, instead it truncates the numerator and denominator in a way that ensures it is lower than a given maximum. So, rather than using the greatest common denominator between the numerator and denominator to reduce the fraction, this `reducedFraction` function merely prunes the numerator and denominator to ensure that they are less than the provided max, and, in so doing, loses precision. In some instances the deviation between the fraction and the result of `reducedFraction` can be over 10%.

Consider either revising the function to reduce the loss of precision or renaming the function to `truncateFraction` and providing more inline documentation that makes the loss of precision clear.

Update: *Fixed in commit* `2cb941a636dd9b0c210dd52ee90907eeb02eea6a` *and commit* `56be8479ec633e0e59385dfcbe66de4759acb4bc`.

Medium Severity

`weightedAverage` calculation might overflow

In the `MathEx` library the function `weightedAverage` performs a multiplication of up to three `uint256` input variables without any prior validation. As the resulting number can have up to `768` bits, it is very likely to lead to an overflow during `uint256`-multiplication.

Currently, the function `weightedAverage` is only used in the `_calcAverageRate` function in the `PoolCollection` contract. Further, the `_calcAverageRate` function is currently used in a way that ensures a maximum size of `128`, `112` and `3` bits for the three operands passed to `weightedAverage`.

including explicit preconditions in the function documentation to ensure that future versions of the codebase remain free from overflows.

Update: *Acknowledged without code changes. In an effort to keep library functions lean, the client generally prefers panics over explicit error messages. Note that the function documentation continues to omit any description about preconditions.*

`calcExpDecayRewards` is at risk of overflow reversions

In the `RewardsMath` contract the `calcExpDecayRewards` function takes `timeElapsed` and `halfLife` values as input, puts them into a `Fraction` struct, and then passes them along to the `MathEx.exp2` function. There is a comment in the inline documentation that

> because the exponentiation function is limited to an input of up to (and excluding) 16 / ln 2, the input value to this function is limited by `timeElapsed / halfLife < 16 / ln 2`

This is problematic because, for any exponential decay distribution program, `halfLife` is fixed while `timeElapsed` will continue to grow after the program is created. For any such program then, it is only a matter of time before `timeElapsed` exceeds `halfLife` by more than the required ratio to overflow a call to `exp2`.

If `timeElapsed / halfLife` exceeds this limit, then the call will simply revert in the `exp2` function with an `Overflow` error. This will cause the entire transaction to revert. Since `calcExpDecayRewards` is called by the `_tokenAmountToDistribute` function, which is called by the `_processRewards` function, no rewards for a program that reaches such a state will be possible.

In such a scenario where the `timeElapsed` is too large, rather than making the call to `exp2` and then reverting, consider having the `calcExpDecayRewards` function return `totalRewards` without any exponentiation calculation. This will result in minimal error (i.e., `1 * totalRewards` instead of `0.9999998874648253 * totalRewards`), but will prevent the case where a program's rewards cannot be distributed because too much time elapses from the program's start.

~~acceptable.~~

Lack of validation

Throughout the codebase, there are places where a proper input/output validation is lacking. In particular:

- The `inverse` and `fromFraction112` functions in `FractionLibrary` do not check if they were provided an invalid fraction.
- The `_mulMod`, `mulDivC`, `mulDivF` functions in the `MathEx` library do not check that `z` in non-zero. In such cases they revert with a Panic, rather than a more helpful error message.
- The `isInRange` and `weightedAverage` functions in the `MathEx` library do not check that the input fractions are valid. The latter also does not ensure it is returning a valid fraction.

A lack of validation on user-controlled parameters may result in erroneous or failing transactions that are difficult to debug. This is especially true in the case of libraries that are capable of being reused across different codebases. To avoid this, consider adding input and output validation to address the concerns raised above.

Update: *Acknowledged without code changes. The client expressed that the lack of reversion on invalid inputs is intentional. In some cases it is desirable behavior to turn divide by zero inputs into zero outputs. In other cases, error messages will be raised somewhere else along the call chain in practice.*

Low Severity

Unbounded number of programs could result in errors

The `programs` function loops over a potentially unbounded list of programs/pools.

Relatedly, the `pools` function requires that all stored values associated with stored pools are first loaded into memory for the call to complete.

In either case, if too many programs are added to the protocol, these functions will run out of gas.

Consider either adding a pagination mechanism to limit the number of stored values that must be loaded or adding some warning to the documentation about the need to keep the number of programs bounded to some safe level to avoid issues with these functions.

Update: *Acknowledged. The client conveyed that simplicity of the external interface was an intentional design choice. If future on-chain usage becomes relevant the respective contract will be upgraded.*

Duplicated code

The internal logic of the `poolTokenAmountToBurn` functions in the `BNTPool` and `PoolCollection` contracts are essentially duplicates.

Duplicating code can lead to issues later in the development lifecycle and leaves the project more prone to the introduction of errors. Such errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical.

Rather than duplicating code, consider having just one contract or library containing the duplicated code and using it whenever the duplicated functionality is required.

Update: *Acknowledged. The flexibility of independent upgrades is prioritized over re-use in this case. Client's response:*

> *If the contracts are used for a while and the shared logic is kept in place without frequent changes, a shared logic component will be created, but for now flexibility is more important than code reusability, especially since these two contracts are very different conceptually.*

`uint32` times can overflow in the future

Throughout the `AutoCompoundingRewards` contract, all program start, end, and elapsed times are encoded as `uint32` values. In the year 2106, this will be problematic in two ways.

1. Near the time when unix timestamps will begin to overflow the `uint32` type, it will become difficult to create flat programs ending after the unix timestamp overflow event. Specifically, it will not be possible to call `createFlatProgram` with an `endTime` that overflows

2. The `createExpDecayProgram` does not expect an `endTime` input. Instead, a `halfLife` input parameter is specified. This will make it possible to call `createExpDecayProgram` even around the time that unix time will overflow the `uint32` type. However, if a program is created before the overflow time, then after the overflow time, even if it should be a valid program, it would be treated as inactive and no rewards from the program could be processed. This, in turn, would prevent both `processRewards` and `autoProcessRewards` from processing rewards for that program. This would be more problematic and tight packing of the stored programs could make an update to remedy the situation more difficult.

Consider either increasing the bits available to store and evaluating timestamps to something larger than `uint32` or adding a warning to the protocol documentation to upgrade the contract well before the overflow time will be reached.

Update: *Acknowledged. Client's response:*

> The contracts are expected to change and evolve and new versions implemented way before the timestamps can overflow (at 2106).

Lack of event emissions after updating state

The following functions do not emit relevant events despite executing state-altering actions:

- No event is emitted when the `_autoProcessRewardsCount` variable is first set during contract initialization. This is inconsistent with the fact that an event *is* emitted when the same value is updated via the `setAutoProcessRewardsCount` function.
- There is no event emitted when `_autoProcessRewardsIndex` is updated despite the fact that this value influences the behavior of the `external` `autoProcessRewards` function.

In order to facilitate tracking and to notify off-chain clients following the contracts' activity, consider emitting events whenever state-changing operations take place.

public implementation detail which should not emit events upon alteration, despite its role in facilitating monitoring.

`enableProgram` is overloaded

In the `AutoCompoundingRewards` contract, the `enableProgram` function is used to enable *or disable* a program for a specific pool, depending on the bool `status` parameter. The function also emits a `ProgramEnabled` event, even in the instance where the program is being disabled (although the event details contain the correct updated status).

In order to avoid unnecessary confusion and to increase the overall readability of the codebase, consider renaming the function and event to something such as `updateProgramStatus` to better reflect the actual function operation.

Update: *Acknowledged. Client's response:*

> This was a design decision in order to prioritize ease of consumption (single event) and binary size over readability.

Notes & Additional Information

Confusing use of `assert` statement

The `assert` statement in the `calcFlatRewards` function of the `RewardsMath` library checks a condition that is impossible only because of how it is used within the `_tokenAmountToDistribute` function. However, given the `calcFlatRewards` function definition and use of the `timeElapsed` and `programDuration` parameters, the purpose of the `assert` statement is not apparent within the `RewardsMath` library.

To enforce input validation within the `calcFlatRewards` function, consider changing the `assert` statement to a `require`, and providing an error message explaining the validation. Alternatively, consider moving the `assert` statement into the `_tokenAmountToDistribute` function to check that the invariant holds in that context.

function still uses `assert` *to check a condition that is not an invariant.*

Unused import

In the `AutoCompoundingRewards` contract the `AccessDenied` import is unused.

Consider removing unused import statements to simplify the codebase and increase overall readability.

Update: *Fixed as of commit* `107b3aa85bc306a0d94705a0bd816e57267bd9f7` *in pull request #430.*

Missing docstrings

Throughout the codebase there are numerous functions missing or lacking documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also 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: *Acknowledged. The comments will be added in future versions of the codebase.*

Conclusions

1 high and 3 medium severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits