PUBLIC

Code Assessment

of the Bancor v3

Smart Contracts

September 27, 2022

Produced for

Bancor

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Bancor Team,

Thank you for trusting us to help Bancor with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Bancor v3 according to Scope to support you in forming an opinion on their security risks.

Bancor implements an AMM exchange protocol with flash loan functionality. The reviewed Bancor v3 tries to mitigate any impairment loss for liquidity providers instantly, has an "Omnipool" for BNT liquidity providers that is used to trade against all other tokens. All tokens can be provided single-sided. In contrast to the previous version, it also has no liquidity caps in the pools.

The most critical subjects covered in our audit were security and functional correctness issues. Most sever is an Oracle Manipulation. All raised issues have been fixed accordingly or were acknowledged by Bancor.

In summary, we find that the codebase provides a good level of security. It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. Especially, for project of this size, they complement but don't replace other vital measures to secure a project.

The communication with the team was always very good and professional. We are happy to receive questions and feedback to improve our service.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	5
• Code Corrected	3
• Risk Accepted	2
Low-Severity Findings	24
• Code Corrected	17
• Specification Changed	1
• Code Partially Corrected	2
• Risk Accepted	3
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Bancor v3 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	22 April 2022	167abb7409ee0853f11062059751d6c755dbe55d	Initial Version
2	28 July 2022	0e14d52706182d6e5934db2e4c030b8981d8bed9	Second Version

For the solidity smart contracts, the compiler version `0.8.13` was chosen.

2.1.1 *Excluded from scope*

The review of any economic principles or business logic is excluded in our technical reviews.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Bancor offers an Automated Market Maker protocol that is composed of one liquidity pool per token - approved by the governance - as well as a single pool (Omnipool) for Bancor's token `BNT`. The pools, compared to alternative AMMs, are single-sided, meaning that liquidity providers can supply a quantity of just one token instead of supplying both tokens of a trading pair. The tokens in Bancor v3's trading pools are always paired with `BNT` and trading between two underlying tokens is realized with 2 hops. This system tries reduces the *impermanent loss* risk for liquidity suppliers. This is done by allowing users to be able to withdraw their provided liquidity in the same token they supplied or, if a pool's liquidity cannot fully satisfy the withdrawal, in an amount of `BNT` equal to the value of the owed tokens (as long as this mechanism is not disabled by the governance). For this reason, pool liquidity and the staking balances of liquidity providers are kept in two different ledgers.

When providing some tokens to the protocol, users receive pool tokens representing their supplied tokens as well as the value accrued on the position due to trading fees. For some token `TKN`, such pool token is named `bnTKN` and follows the `ERC20` token standard. If a user supplies `BNT`, they receive an equal amount of `vBNT`, the governance token of the protocol, additionally to their `bnBNT`.

To implement impermanent loss protection, an exit fee is deduced on withdrawal and a cooldown period is enforced. During this duration, a user must stake their pool tokens and does not benefit from any additionally accrued interest. Furthermore, Bancor v3 employs a special withdrawal algorithm that opens small arbitrage opportunities in order to offset the impermanent loss risk of the protocol. At the time of this writing, the arbitrage mechanism has been partially disabled.

Newly created pools are open for deposits but do not allow any trading. If a certain threshold is reached, governance can enable trading on the pools with a fixed price. This will bootstrap the pool with a pre-determined amount of liquidity for which `BNT` are minted. This liquidity is allowed to grow on certain actions such as depositing if there are more underlying tokens available in the pool. It is capped by a maximum that is set by the governance for each pool.

The protocol keeps `AverageRates`, which are weighted arithmetic means for both the price and the inverse of the price. It is used to make sure that certain actions can only be performed when a pool's rate is stable to prevent manipulations.

2.2.1 BancorNetwork

The contract is the main entry point for most of the interactions with the project.

- `deposit` / `depositFor` are used to provide some tokens to the protocol and obtain some corresponding pool tokens in exchange.
- `initWithdrawal` initializes a withdrawal. The pool tokens of the user are sent to the `PendingWithdrawals` contract which makes sure that the final withdrawal can only take place after the aforementioned cooldown period.
- `cancelWithdrawal` can be called during the withdrawal cooldown period to get back the provided pool tokens that were sent to the `PendingWithdrawals` contract.
- `withdraw` can be called after the withdraw cooldown period. It will process the withdrawal of the user by sending the amount of tokens owed. If the protocol does not hold enough tokens to satisfy the request, the rest is compensated by minting an amount of `BNT` of equal value. When withdrawing BNT, in addition to pool tokens, `vBNT` must be sent back to the protocol.
- `tradeBySourceAmount` / `tradeByTargetAmount` are used to perform an actual trade either between some token and `BNT` or between two different base tokens. In the latter case, a multihop trade between two pools is executed. Additionally, a fee is deduced of which one part is earned by the liquidity providers and another part is swapped back to `BNT` (if necessary) and added to the ledger that allows the `VortexBurner` to buy back `vBNT` and burn them.
- `flashLoan` can be used to loan an arbitrary amount of tokens as long as they are returned in the same transaction.
- `migratePools` can be called to migrate a pool from a newer `PoolCollection`, if one exists.

The following functions can be only called by the admin or some privileged roles:

- `registerPoolCollection` adds a new `PoolCollection` to the network. This is only allowed if no other `PoolCollection` with the same type and version have already been registered.
- `unregisterPoolCollection` removes the given pool collection from the network.
- `createPool` creates a pool for the given token.
- `createPools` creates multiple pools for the supplied list of tokens.
- `migrateLiquidity` can be called by the migration manager to migrate user's funds into the Bancor protocol. This is essentially a `deposit` function that accounts for some special use cases.
- `withdrawNetworkFees` can be called by the network fee manager (`BancorVortex`) to withdraw fees generated from trades.
- `pause` and `resume` can only be called by the emergency stopper and will respectively pause or resume users' interactions with the contract. Functions that are restricted to the governance are not affected by this toggle.
- `enableDepositing` is called by the governance to enable or disable deposits in all pools.

2.2.2 PoolCollection

Pools are handled by pool collections. The system is intended for multiple different types of collections. As of this writing, only one type exists. Pool collections are immutable (i.e., cannot be upgraded). If a pool collection has to be changed, a new version is deployed, and pools can be migrated from old collections to the new one. Only after all pools have been migrated, a pool collection can be deleted from `BancorNetwork`.

Pool collections handle the liquidity of certain pools and can grow / shrink them based on `deposit` / `withdraw` actions. A pool's liquidity is only changed when the spot price does not diverge by more than a pre-set percentage from the `AverageRate`.

The following function are restricted to the governance:

- `setDefaultTradingFeePPM` can be called to update the default trading fee.
- `setTradingFeePPM` can be called to update the trading fee of a given pool.
- `enableTrading` and `disableTrading` respectively enable or disable trading for a given pool.
- `enableDepositing` enable or disable depositing for a given pool.
- `updateTradingLiquidity` is used to manually update the trading liquidity of a pool.

The following functions can only be called by the `BancorNetwork` contract.

- `createPool`, `depositFor`, `withdraw`, `tradeBySourceAmount` `tradeByTargetAmount` are called by the respective functions in `BancorNetwork`.
- `onFeesCollected` can be called to increase the staked balance when fees are collected.

The following functions are only callable by the Pool Migrator.

- `migratePoolIn`
- `migratePoolOut`

2.2.3 BNTPool

The `BNTPool` holds the ledger for all staked `BNT` and is used to mint and burn `BNT` according to the liquidity needs of the pools. Contrary to base tokens, only a finite amount of `bnBNT` is available. If a user provides liquidity to the pool, their sent `BNT` are burned, and the user receives some of the already minted `bnBNT` that are owned by the protocol.

The following functions can be called by `PoolCollection`:

- `mint` is used in withdrawals to mint `BNT` as compensation if there is not enough base token liquidity available. Additionally, it is called to add additional `BNT` liquidity to a pool.
- `burnFromVault` is used to remove excess `BNT` liquidity from a pool.
- `requestFunding` is used to add `BNT` liquidity along with `bnBNT` to the pool if the liquidity is allowed to grow due to a stable rate.
- `renounceFunding` is used to remove `BNT` liquidity along with `bnBNT` from the pool if liquidity is reset or if a withdrawal does not use the in-built arbitrage mechanism.

The following functions can only be called by the `BancorNetwork` contract:

- `depositFor` and `withdraw` are called by the respective homonymous function of `BancorNetwork`.
- `onFeesCollected` can be called to increase the `BNT` staked balance when fees are collected.

2.2.4 BancorV1Migration

This contract allows users to withdraw supplied liquidity from Bancor's legacy contracts to Bancor v3.

2.2.5 NetworkSettings

This contract holds all the settings of the protocol. For example, the minimum liquidity a pool must contain for trading to be enabled or the fee perceived by the protocol on different actions such as flash loans, withdrawals, or trades. The values can be changed by the governance.

2.2.6 PendingWithdrawals

This contract is used to manage withdrawals.

- `setLockDuration` is only callable by the admin to change the cooldown period.

Its three other main functions `initWithdrawal`, `cancelWithdrawal` and `completeWithdraw` are only callable by the `BancorNetwork` contract. The two former functions are called by the respective homonymous functions of `BancorNetwork`. `completeWithdrawal` is called by `BancorNetwork.withdraw` and returns the held pool tokens of a withdrawal request back to the network for further use.

2.2.7 PoolMigrator

This contract is called by `BancorNetwork.migratePools` and hosts the logic for migrations of single or multiple pools between pool collections.

2.2.8 PoolToken

This contract implements a pool token. It is an `ERC20` token which is both ownable and burnable. It additionally holds the address of the token it represents.

2.2.9 PoolTokenFactory

This factory is used to create pool token contracts. Its main function `createPoolToken` creates a pool token with the right name, symbol and decimals and transfers its ownership to the caller.

2.2.10 AutoCompoundingRewards

The auto-compounding reward mechanism distributes value to all pool token holders by burning a certain amount of staked pool tokens over time. As the staked balance remains constant (or grows), burning pool tokens automatically increases the value of the remaining pool tokens. Thus, no further actions from liquidity providers are required in order to benefit from reward programs set up with this contract.

The following functions can be called by the governance:

- `createFlatProgram` and `createExpDecayProgram` can be called to create new programs. Prior to this call, the pool tokens that will be burned for this program have to be sent to the `ExternalRewardsVault`.
- `terminateProgram` can be called to terminate a given program.
- `enableProgram` is used to enable or disable a program.

The following functions can be called by anyone:

- `processRewards` computes how much tokens have to be distributed since its last call and burn the corresponding amount of pool tokens.
- `autoProcessRewards` performs `processRewards` in round-robin manner for a specific amount of programs.

2.2.11 StandardRewards

As the auto-compounding reward mechanism can only be used to distribute value in the same token pool token holders have staked, the standard reward mechanism allows for rewards in BNT. Users must manually stake their pool tokens in the contract to receive rewards.

The following functions can be called by the governance:

- `createProgram` can be called to create a new program.
- `terminateProgram` can be called to terminate a given program.
- `enableProgram` is used to enable or disable a program.

The following functions can be called by anyone:

- `join` and `joinPermitted` are used to join a program by staking some pool tokens.
- `leave` allows users to unstake their pool tokens.
- `depositAndJoin` is a helper function that will first deposit tokens in Bancor v3 and then stake the obtained pool tokens into the contract.
- `claimRewards` allows users to receive their rewards from the last time they claimed up until the call.
- `stakeRewards` can be called to deposit all accrued rewards into Bancor v3.

2.2.12 BancorPortal

This contract is the entry point for users wanting to migrate their Uniswap v2 or SushiSwap v2 positions into Bancor v3 easily. Its two main functions `migrateUniswapV2Position` and `migrateSushiSwapPosition` will close user's liquidity positions on the respective protocols and reinvest them into Bancor v3.

2.2.13 Vaults

The protocol uses three different vaults. Each vault accepts native `ETH` and `ERC20` deposits. The main pool contracts are allowed to withdraw funds from the vaults.

- The `MasterVault` is where all the pools' liquidities are stored. It hence holds both `BNT` and the different whitelisted tokens.
- The `ExternalRewardVault` is used to fund the different reward programs; it can hold any reward token which will either be burned or distributed depending on the type of program they are funding.
- The `ExternalProtectionVault` is funded by token projects in order to gain approval of the governance to be listed in Bancor v3. Since the impermanent loss protection of the protocol imposes certain risks, particularly for tokens that increase in value faster than the overall market, extra funds in this wallet or used to protect the protocol and distributed to liquidity suppliers before any `BNT` are minted as compensation.

2.2.14 Role overview and trust model

The system has the following explicitly and implicitly defined roles. We assume the described trust level for each of the roles:

- Deployer: Each contract has a deployer that needs to correctly set the initial state of the contract. The initial state is e.g., immutable addresses of the Bancor v3 ecosystem. Most contracts use a proxy scheme. The proxy deployer is assigned the `ROLE_ADMIN` which is the super admin of all other roles.
- `ROLE_ASSET_MANAGER`: Three contracts implement this role. The role has the power to call `withdrawFunds` or `burn` on vault contract (except BNT vault). Both functions are critical functions

and, hence, the role needs to be fully trusted. For the ExternalProtectionVault and the MasterVault the poolCollectionAddress is granted this role in the code automatically. The ROLE_ASSET_MANAGER for the ExternalRewardsVault needs the role to be set by the ROLE_ADMIN. StandardRewards and BancorNetwork also need to have the role ROLE_ASSET_MANAGER set by the ROLE_ADMIN to call withdrawFunds and burn at the corresponding contracts.

- ROLE_BNT_POOL_TOKEN_MANAGER: This role is only defined in the BNTPool contract. This role is the ROLE_ASSET_MANAGER for the BNT vault tokens (bnBNT). The role is separated from all other tokens that are managed by the ROLE_ASSET_MANAGER but effectively has the same power but for BNT.
- ROLE_BNT_MANAGER: This role is required to mint BNT in the BNTPool contract and to act like the ROLE_ASSET_MANAGER and ROLE_BNT_POOL_TOKEN_MANAGER role to withdraw and burn from the MasterVault. The role is automatically granted to the poolcollection address for the BNTPool and needs to be manually set for the MasterVault. The role needs to be fully trusted like the similar roles above as it has similar power and can additionally mint BNT.
- ROLE_VAULT_MANAGER: the vault manager role is required to request the BNT pool to burn BNT from the master vault. The role is only defined in the BNTPool and anyone having it can call burnFromVault with an arbitrary amount of bnt to burn from the masterVault. We assume hence that this role is fully trusted. In the current implementation, it is granted to the pool collections added to the network.
- ROLE_FUNDING_MANAGER: the funding manager role is required to request or renounce funding from the BNT pool. It is only defined in the BNTPool and allows calling requestFunding and renounceFunding. As these functions respectively mint and burn both BNT and bnBNT from the protocol and master vault, this role is considered as fully trusted. It is currently granted to the pool collections added to the network.
- ROLE_MIGRATION_MANAGER: the migration manager role is required for migrating liquidity. It is only used in the BancorNetwork contract and allows calling migrateLiquidity, as someone could call this function with an arbitrary value originalAmount leading more or less vBNT being minted by BNTPool.depositFor, we consider this function as fully trusted.
- ROLE_EMERGENCY_STOPPER: the emergency manager role is required to pause/unpause the network. It is defined in the BancorNetwork contract and allows calling pause and resume. These two functions prevent or allow access to the main functionalities of the protocols. As this role would allow a malicious holder to perform denial of services, it is considered as fully trusted.
- ROLE_NETWORK_FEE_MANAGER: the network fee manager role is required to pull the accumulated pending network fees. As is used in the BancorNetwork contract and allows withdrawing the network fees of the protocol by calling withdrawNetworkFees, this role is fully trusted.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	2

- Incorrect Liquidity Decrease Risk Accepted
- Missing Slippage Protection Risk Accepted

Low-Severity Findings	6

- Missing Getter for Average Rates Acknowledged
- Fake Pool Token Migration Risk Accepted
- Gas Savings Code Partially Corrected
- Inconsistent Reentrancy Protection Code Partially Corrected
- Rounding Error Can Lock Tokens Risk Accepted
- BNTPool.renounceFunding Division by 0 Risk Accepted

5.1 Incorrect Liquidity Decrease

Design Medium Version 2 Risk Accepted

`PoolCollection._calcTargetTradingLiquidity` decreases the BNT trading liquidity if the current funding of a certain pool is greater than its funding limit. This is done in a way that could possibly reset the pool:

```
uint256 excessFunding = currentFunding - fundingLimit;
targetBNTTradingLiquidity = MathEx.subMax0(liquidity.bntTradingLiquidity, excessFunding);
```

Consider the following example:

- The funding limit is 40,000 BNT.
- The current funding of the pool is 40,000 BNT.
- `bntTradingLiquidity` is 20,000 BNT (for example after the value of BNT to the corresponding token has quadrupled).
- The funding limit is now lowered to 20,000 BNT by governance.
- `bntTradingLiquidity` is now set to 0 and the pool is reset on the next deposit.

Risk accepted

Bancor plans to fix this issue in a future version and accepts the risk for now.

5.2 Missing Slippage Protection

Security Medium Version 1 Risk Accepted

The following functions do not guarantee any slippage protection for users and are thus susceptible for front-running attacks:

- `BancorPortal._uniV2RemoveLiquidity` calls the functions `removeLiqudity` and `removeLiquidityETH` of `UniswapV2Router02` with 1 wei slippage protection at all circumstances.
- `BancorV1Migration.migratePoolTokens` calls `removeLiquidity` in Bancor v1's `StandardPoolConverter` with 1 wei slippage protection at all circumstances.

Risk accepted:

The client accepts the risk, stating the following:

> Similar to how liquidity removal is processed on these 3rd party protocols, it is assumed that users will migrate their liquidity immediately and will be prompted with its results.

5.3 Missing Getter for Average Rates

Design Low Version 2 Acknowledged

The function `PoolCoollection.poolData` is commented as follows:

> there is no guarantee that this function will remain forward compatible, so relying on it should be avoided and instead, rely on specific getters from the IPoolCollection interface

This indicates that all data from the struct that is returned by this function is also available via independent getters. There is, however, no getter function available that returns the `AverageRates`.

Acknowledged

Bancor acknowledged the issue and plans to fix it in a future version.

5.4 Fake Pool Token Migration

Security Low Version 1 Risk Accepted

`BancorV1Migration.migratePoolTokens` does not check if the given pool token is registered in the `ContractRegistry`. For this reason, an attacker could call the function with a fake pool token contract that returns a fake `StandardPoolConverter`:

```
IBancorConverterV1 converter = IBancorConverterV1(payable(poolToken.owner()));
```

This converter can then in turn return reserve amounts of tokens that do not exist:

```
uint256[] memory reserveAmounts = converter.removeLiquidity(amount, reserveTokens, minReturnAmounts);
```

If the `BancorV1Migration` contract holds only tokens for some reason, these tokens can then be sent to the attacker as the contract does not check its token balances before calling `converter.removeLiquidity`.

Risk accepted:

The client accepts the risk noting that the contract is not supposed to receive any tokens.

5.5 Gas Savings

Design | Low | Version 1 | Code Partially Corrected

The following list contains suggestions on how the gas consumption of Bancor v3 can be improved:

- The fields of `MigrationResult` in `BancorPortal` can be rearranged to achieve tighter packing.
- `BancorNetwork.addPoolCollection` loads the latest pool collection from storage then calls `_setLatestPoolCollection` which loads the same pool collection from storage again.
- `BancorNetwork.createPools` loads the same pool collection on each iteration from storage.
- `BancorNetwork._depositBNTFor` transfers BNT to the `BNTPool` which then burns them. The tokens could be burned by `BancorNetwork` instead.
- `BancorNetwork.withdraw` transfers pool tokens from `PendingWithdrawal` to `BancorNetwork`. `_withdrawBNT` then approves `BNTPool` for the tokens and transfers the tokens to `BNTPool`. The tokens could be directly transferred from `PendingWithdrawals` to `BNTPool`.
- `BancorNetwork._withdrawBNT` transfers vBNT from the provider to `BNTPool` which then burns them. The tokens could be burned directly from the provider's address.
- `BancorNetwork.withdraw` transfers pool tokens from `PendingWithdrawal` to `BancorNetwork`. `_withdrawBaseToken` then approves `PoolCollection` for the tokens which burns them. The tokens could be directly burned from `PendingWithdrawals`.
- Some or all fields in `NetworkSettings` could be immutable and updated via proxy upgrade, depending on how frequently they are updated and loaded.
- Many call chains unnecessarily validate input data more than once. For example, `BancorNetwork.initWithdrawal` validates the pool token address and amount, then calls `PendingWithdrawals.initWithdrawal` which performs the same validations again even though it can only be called by the `BancorNetwork` contract.
- `PoolCollection.depositFor` loads `data.liquidity.stakedBalance` from storage and then loads the whole `data.liquidity` struct from storage with no change of the data in-between.
- `PoolCollection.depositFor` reads `data.liquidity` from storage, updates the fields on storage and then loads the struct again two times from storage.
- `PoolMigrator.migratePool` retrieves the target pool collection of a given pool by calling `BancorNetwork.latestPoolCollection`. It then calls `PoolCollection.migratePoolOut` which performs the same call to check if the given target pool collection is valid.
- `PoolCollection._poolWithdrawalAmounts` takes the `Pool` struct as `memory` copy. Since not all words are accessed and the function is only called with storage pointers, the `data` argument could be a storage pointer and the necessary fields could be cached inside the function.

- `PoolCollection._executeWithdrawal` loads `stakedBalance` from storage even though it is already cached in `prevLiquidity`
- `PoolCollection._updateTradingLiquidity` calls `_resetTradingLiquidity` which loads `liquidity.bntTradingLiquidity` from storage even though an overloaded version of the function exists which takes that variable as an argument and there is a cached version of `liquidity` available.
- `PoolCollection._performTrade` loads the `liquidity` struct from storage even though the values are already present in the `TradeIntermediateResult` argument.
- `PoolMigrator._migrateFromV1` translates the `Pool` struct from the old version to the new version. Since the structs are identical, this is not necessary.
- `PoolToken._decimals` can be immutable.
- Because pools can only be added by the admin address, `PoolTokenFactory.createPoolToken` could take the override variables as arguments instead of using storage variables.
- `AutoCompoundingRewards.terminateProgram` loads `ProgramData` from storage to check if the given program exists. As the pool is later removed from `_pools`, the call would revert anyways if the the pool program did not exist.
- In `AutoCompoundingRewards` some functions (e.g. `enableProgram`) load the whole `ProgramData` struct from storage even though not all words are required.
- In `StandardRewards` some functions (e.g. `createProgram`) load the whole `ProgramData` struct from storage even though not all words are required.
- The fields `_bntPool`, `_pendingWithdrawals` and `__poolMigrator` in `BancorNetwork` could be `immutable` if they are set up either directly in the constructor of `BancorNetwork` or with pre-known addresses.
- In `BancorNetwork.flashloan`, the user could pay the loaned amount directly back to the master vault.
- During a withdrawal of base tokens (not BNT), all pool tokens could be burned in `PendingWithdrawals` instead of burning a part of them, sending the rest to `BancorNetwork` and finally burning them in `PoolCollection`.

Code partially corrected:

The client has addressed some of the suggestions. Additionally, some are no longer relevant due to other code changes.

- **Corrected:** The fields of `MigrationResult` in `BancorPortal` are now tightly packed.
- **Corrected:** `latestPoolCollection` has been completely removed from the code.
- **Corrected:** `BancorNetwork.createPools` takes the respective pool collection as argument.
- **Not corrected:** `BancorNetwork._depositBNTFor` still transfers BNT to the `BNTPool` which then burns them.
- **Corrected:** `BancorNetwork._withdrawBNT` directly transfers pool tokens from `PendingWithdrawal` to `BNTPool`.
- **Not corrected:** `BancorNetwork._withdrawBNT` still transfers vBNT from the provider to `BNTPool` which then burns them.
- **Partially corrected:** `BancorNetwork._withdrawBaseToken` directly transfers pool tokens from `PendingWithdrawal` to `PoolCollection`.
- **Not corrected:** All fields in `NetworkSettings` are still storage variables.

- **Not corrected:** There are still many call chains that validate input multiple times.
- **Not corrected:** `PoolCollection.depositFor` still redundantly loads `data.liquidity.stakedBalance` from storage.
- **Not corrected:** `PoolCollection.depositFor` still redundantly loads `data.liquidity` multiple times from storage.
- **Corrected:** `latestPoolCollection` has been completely removed from the code.
- **Corrected:** `PoolCollection._poolWithdrawalAmounts` takes only relevant and cached data as input.
- **Not corrected:** `PoolCollection._executeWithdrawal` still loads `stakedBalance` from storage.
- **Corrected:** `PoolCollection._updateTradingLiquidity` uses the overloaded version of `_resetTradingLiquidity`.
- **Not corrected:** `PoolCollection._performTrade` still loads the `liquidity` struct from storage.
- **Not corrected:** `PoolMigrator._migrateFromV1` has been renamed to `PoolMigrator._migrateFromV5` but still unnecessarily translates equal structs.
- **Not corrected:** `PoolToken._decimals` is still a storage variable.
- **Not corrected:** `PoolTokenFactory.createPoolToken` still uses storage for override variables.
- **Not corrected:** `AutoCompoundingRewards.terminateProgram` still redundantly checks for pool existance.
- **Not corrected:** In `AutoCompoundingRewards` some functions (e.g. `pauseProgram`) still load more data from storage than required.
- **Not corrected:** In `StandardRewards` some functions (e.g. `_programExists`) still load more data from storage than required.
- **Not corrected:** The fields `_bntPool`, `_pendingWithdrawals` and `__poolMigrator` are still stored in storage.
- **Not corrected:** In `BancorNetwork.flashloan`, the loaned amount is still paid back to the contract and then sent to the master vault.
- **Corrected:** `PendingWithdrawals.completeWithdrawal` does not burn tokens anymore.

5.6 Inconsistent Reentrancy Protection

Design | Low | Version 1 | Code Partially Corrected

- `AutoCompoundingRewards.createProgram` has a reentrancy protection, while the functions `terminateProgram` and `enableProgram` do not.
- `BancorNetwork.withdrawNetworkFees` does not have reentrancy protection, while other functions that are restricted to callers with certain roles have.

Code partially corrected

Although all mentioned functions are now protected against reentrancy, the new function `AutoCompoundingRewards.setAutoProcessRewardsCount` lacks reentrancy protection while all other functions restricted to the admin are protected.

5.7 Rounding Error Can Lock Tokens

Design Low Version 1 Risk Accepted

When creating a `StandardRewards` program, both the reward rate and the remaining rewards are computed as follows:

```
uint256 rewardRate = totalRewards / (endTime - startTime);
_programs[id] = ProgramData({
    ...
    rewardRate: rewardRate,
    remainingRewards: rewardRate * (endTime - startTime)
});
```

Depending on the token's number of decimals and the duration of the program, `remainingRewards` can be smaller than `totalRewards` due to the divide-then-multiply scheme. In practice, this means that in such cases, `totalRewards - remainingRewards` tokens won't be distributed and will never be deduced from `_unclaimedRewards`. This results in the tokens being locked in the contract and not being able to be used for future programs.

Risk accepted:

The client accepts the risk, stating the following:

> The amount of tokens which can be locked due to a rounding error is negligible. We are also considering revamping the whole mechanism, which will also affect this code.

5.8 `BNTPool.renounceFunding` Division by 0

Correctness Low Version 1 Risk Accepted

In `PoolCollection._executeWithdrawal`, if the protocol has to renounce BNT funding and this results in the BNT staked balance being reset to 0, but the BNT trading liquidity is still greater than 0, `_resetTradingLiquidity` is called which tries to renounce BNT funding again. As the staked balance has already been set to 0, this second call to `BNTPool.renounceFunding` will now revert due to a division by 0.

Consider the following case:

- User A deposits TKN liquidity into an empty pool.
- Trading is enabled.
- User B trades a certain amount of TKN for BNT.
- User A now withdraws all his supplied TKN.
- The withdrawal fails due to the mentioned problem.

Risk accepted:

The client accepts the risk, stating the following:

> This is a rare case that we don't expect to happen in practice, since trading can't be enabled immediately and usually involves many depositors. In any case, we will consider addressing this in the future as well.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	1

- Oracle Manipulation Code Corrected

Medium-Severity Findings	3

- BNT Burned Twice Code Corrected
- Locked vBNT Code Corrected
- Pool Denial of Service Code Corrected

Low-Severity Findings	18

- BNT Deposit Allows msg.value > 0 Code Corrected
- Consistency Issues Between Implementation, Excel Demonstration and Documentation Regarding the Withdrawal Specification Changed
- Different Programs Can Share the Same Reward Code Corrected
- Emission of Events With Arbitrary Amounts Code Corrected
- Impossible to Migrate ETH Position Code Corrected
- Inconsistent Naming Code Corrected
- Inconsistent Use of ERC20.transfer Code Corrected
- Misleading Comment Code Corrected
- Misleading Comment in PoolCollection.enableTrading Code Corrected
- Problematic Loop Continuation During Pool Migration Code Corrected
- Undocumented Behavior Code Corrected
- Unused Imports / Variables Code Corrected
- Wrong Function Name in BancorPortal Code Corrected
- Wrong Interface Code Corrected
- AutoCompoundingRewards Can Burn More Pool Tokens Than Expected Code Corrected
- BNTPool.renounceFunding Fails on Insufficient BNT Pool Token Balance Code Corrected
- ERC20Permit Handling Code Corrected
- MathEx.reducedFraction Can Turn Denominator to 0 Code Corrected

6.1 Oracle Manipulation

Security High Version 1 Code Corrected

To prevent manipulation, Bancor v3 calculates a moving average of each pool's spot price that is adjusted once per block. Critical actions like the increase of trading liquidity or withdrawal of funds requires the spot rate of a pool to not diverge from this moving average by more than a certain percentage.

Since the moving average is calculated as an arithmetic mean, it is subject to manipulation. Consider the following scenario:

- An attacker funds a pool with some tokens with a spot rate of 1 BNT : 1 token.
- They perform a trade from BNT to the token by providing an amount of BNT that changes the spot rate to 10 BNT : 1 token. The average rate is now 2.8 BNT : 1 token.
- In the next block, they perform another trade from token to BNT to bring the spot rate back to 2.8 BNT : 1 token.
- Since the spot rate now equals the average rate, the attacker can withdraw his supplied tokens.
- The pool does not contain enough tokens to satisfy the withdrawal, so the attacker gets compensated in BNT for the outstanding amount.
- This compensation is calculated with the average rate of the pool which now is 2.8 BNT to 1 token instead of the real 1 : 1 rate.
- The attacker will receive 2.8 times the amount of BNT he is actually eligible to receive.

The attacker is required to split both trades in 2 consecutive blocks. In the first block, they create an arbitrage opportunity that can be utilized by an arbitrageur. To make sure, their initial investment will not be lost, they must selfishly mine 2 blocks in a row. This is possible with around 1.5% of the total hashrate of Ethereum. Renting this amount of hashrate is in the realm of possibilities and we estimate that the cost of renting the hashrate and losing out on the reward of the additional mined blocks results in a total cost of ~150.000 USD.

Alternatively, an attacker could try to spam transactions to the Ethereum network in order for their second transaction to be executed before the transactions of any arbitrage bot.

Furthermore, after Ethereum's transition to Proof-of-Stake, the attack becomes simpler: As the attacker now knows when it is their turn for validation, they could submit their first transaction right to the block before. Using Flashbots, the transaction could actually be hidden so that no arbitrage bots would see it before it is included in the block. The next block is then in the hand of the attacker.

While this attack is hard to carry out and requires a lot of capital, it can also create immense losses.

Code corrected:

A second moving average for the inverse rate has been introduced. Averages for the rate and the inverse rate are calculated independently which prevents the aforementioned attack. The resulting inverse rate in the example would diverge from the inverse spot rate by ~100%.

6.2 BNT Burned Twice

Correctness | Medium | Version 1 | Code Corrected

Withdrawals from `PoolCollection` can result in the burning of double the amount of BNT than intended. This happens any time a withdrawal occurs that results in the protocol removing BNT from the protocol equity. In this case, both `amounts.bntProtocolHoldingsDelta.value` and `amounts.bntTradingLiquidityDelta.value` are set to the same value greater than 0, resulting in a call to `BNTPool.renounceFunding` which burns the amount of BNT and a call to `BNTPool.burnFromVault` which burns the same amount of BNT again.

Code corrected:

If both `amounts.bntTradingLiquidityDelta.value` and `amounts.bntTradingLiquidityDelta.value` are greater than 0, only the former value triggers token burning (via `BNTPool.renounceFunding`).

6.3 Locked vBNT

Design | Medium | Version 1 | Code Corrected

In the `StandardRewards` contracts, users can call `depositAndJoin` or `depositAndJoinPermitted` to deposit underlying tokens and stake the obtained pool tokens in one single transaction. To perform such aggregation, the protocol transfers the tokens from the user to itself and calls `BancorNetwork.depositFor` to get the pool tokens that will then be used for staking.

If the token being deposited is `BNT`, `BancorNetwork` will send both `bnBNT` and `vBNT` to the contract. As there is no handling for `vBNT`, it will stay locked into the contract, preventing the user to ever withdraw his `BNT` from the network.

Code corrected:

`depositAndJoin` now keeps the pool tokens, but sends vBNT back to the provider if BNT are deposited. Additionally, a temporary function `transferProviderVBNT` has been added to allow distribution of already accumulated vBNT to their owners by the contract admin.

6.4 Pool Denial of Service

Security | Medium | Version 1 | Code Corrected

The first user to deposit into a newly created pool can immediately burn his pool tokens. In this case, depositing into the pool no longer works, because the following check will revert:

```
if (poolTokenSupply == 0) {
    if (stakedBalance > 0) {
        revert InvalidStakedBalance();
    }
}
```

A malicious user could create a bot that performs this attack cheaply by instantly depositing 1 wei base tokens into any newly created pool.

Code corrected:

New deposits now reset the pool when pool token supply is 0 and staked balance is greater than 0.

6.5 BNT Deposit Allows `msg.value > 0`

Correctness | Low | Version 1 | Code Corrected

`BancorNetwork.depositFor` is `payable`. While the `_depositBaseTokenFor` function makes sure that it reverts if the sent token is not equal to ETH and `msg.value` is greater than 0, the same check is not applied in `_depositBNTFor`.

Code corrected:

`_depositBNTFor` now reverts if `msg.value` is greater than 0.

6.6 Consistency Issues Between Implementation, Excel Demonstration and Documentation Regarding the Withdrawal

Correctness | Low | Version 1 | Specification Changed

1. **Deviating Calculation of** `BNT` **Burned**

The formula for `BNT` trading liquidity to be burned from the pool defined in the excel spreadsheet `renounce nothing method.A11` is deviating from the white paper and the implementation. It seems like the sign in front of `B10*E10` needs to be removed.

The documented and implemented formula is:

$$P = \frac{ax(b+c+e(2-n))}{(1-m)(be+x(b+c-e(1-n)))}$$

The formula used in excel is:

$$P = \frac{ax(b+c+e(2-n))}{(1-m)(-be+x(b+c-e(1-n)))}$$

2. **Incorrect Denominator**

The documentation on page 39 has the following formula documented for $hmax_{supr}$

$$hmax_{surp} = \frac{be(en+m(b+c-e))}{(1-m)(b+c-e(1-n))}$$

The formula's denominator is missing the additional term `(b+c-e)`

Specification changed

Bancor replied the following:

> Both issues outlined here were actually typing errors in the spec, while the implementation is correct. The spec was updated.

6.7 Different Programs Can Share the Same Reward

Correctness | Low | Version 1 | Code Corrected

`AutoCompoundingRewards` ensures that only one program exists for a specific pool at a given time. In practice, this means that the pool tokens allocated for such programs cannot be used by another one.

Similarly, using `_unclaimedRewards`, `StandardRewards` makes sure that if multiple programs have the same reward token, the external reward vault must contain enough tokens to cover all of them.

However, if `StandardRewards` and `AutoCompoundingRewards` contain programs for the same reward token and the address for the `_externalRewardsVault` is the same in both contracts, correct funding cannot be ensured because both programs only check that there are enough funds in the vault.

Code corrected:

`StandardRewards` now only distributes BNT via minting. It does not access the `_externalRewardsVault` anymore.

6.8 Emission of Events With Arbitrary Amounts

Design | Low | Version 1 | Code Corrected

`BancorNetwork._initWithdrawal` does not check if the supplied pool token address belongs to a pool. An attacker could call the function with a fake pool token that returns a valid pool address:

```
Token pool = poolToken.reserveToken();
if (!_network.isPoolValid(pool)) {
    revert InvalidPool();
}
```

The contract would then transfer the fake pool tokens from the attacker while the attacker keeps the real pool tokens and emit a `WithdrawalInitiated` event with arbitrary pool token amounts. While this is not a problem for the protocol itself and is also not exploitable in the final withdrawal, third party applications relying on the emitted events could be affected.

Code corrected:

`_initWithdrawal` now correctly checks if the supplied pool token is valid.

6.9 Impossible to Migrate ETH Position

Design | Low | Version 1 | Code Corrected

When trying to migrate a Uniswap or SushiSwap position, if one of the tokens is the protocol defined native token address `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`, the call to the factory to obtain the pair's address will return the address zero and the transaction will revert with `NoPairForTokens` as this custom address is not used in these protocols.

Code corrected

Bancor now interacts with Uniswap and SushiSwap using the WETH address instead of `0xEee...EEeE`.

6.10 Inconsistent Naming

Design | Low | Version 1 | Code Corrected

`BancorPortal.migrateSushiSwapV1Position` returns a struct with "Uniswap" in one of its field's names.

Code corrected:

`migrateUniswapV2Position` and `migrateSushiSwapPosition` now both return a struct with the name `PositionMigration`.

6.11 Inconsistent Use of `ERC20.transfer`

Design | Low | Version 1 | Code Corrected

In some cases a normal transfer is used. Some of these occurrences have the comment:

```
// transfer the tokens to the provider (we aren't using safeTransfer, since the PoolToken is a fully
// compliant ERC20 token contract)
p.poolToken.transfer(provider, poolTokenAmount)
```

But the pool token is also transferred with a safe transfer in another case

```
poolToken.safeTransferFrom(provider, address(_pendingWithdrawals), poolTokenAmount)
```

The assumption that all tokens behave as expected should be carefully evaluated against gas savings between a normal transfer and a safe transfer.

Code corrected:

`BancorNetwork._initWithdrawal` now transfers the pool tokens using a regular `transferFrom` call. Since all pool tokens are `PoolToken` contracts, they revert on failure making it safe to use the regular `transferFrom` function.

6.12 Misleading Comment

Design | Low | Version 1 | Code Corrected

`_latestPoolCollections` in `BancorNetwork` has the following comment:

> a mapping between the last pool collection that was added to the pool collections set and its type

Since the function `setLatestPoolCollection` allows the governance to set to latest pool collection to any pool collection, the comment is incorrect.

Furthermore, when the the "latest" pool collection is set to an older version, multiple pool collections with the same version can be added through `addPoolCollection`.

Code corrected:

The `latestPoolCollections` mechanism has been completely removed.

6.13 Misleading Comment in PoolCollection.enableTrading

Design | Low | Version 1 | Code Corrected

A comment of `PoolCollection.enableTrading` states:

> if the price of one (10**18 wei) BNT is $X and the price of one (10**6 wei) USDC is $Y, then the virtual balances should represent a ratio of X to Y*10**12

The explanation is ambiguous and could be misunderstood in a way that both virtual balances must be represented with the same number of decimals.

Code corrected:

The addressed documentation has been improved to clarify possible misunderstandings.

6.14 Problematic Loop Continuation During Pool Migration

Correctness | Low | Version 1 | Code Corrected

`BancorNetwork.migratePools` checks if `newPoolCollection` is equal to the 0-address. In the current setup this can never happen.

Furthermore, if the `PoolMigrator` implementation changes for future `PoolCollection` versions so that the `migratePool` function could indeed return the 0-address, the mentioned check leads to a continuation of the migration loop:

```
if (newPoolCollection == IPoolCollection(address(0))) {
    continue;
}
```

In this case, the pool data of the old pool would be lost and `_collectionByPool` would point to a pool collection that does not contain the migrated pool anymore.

Code corrected:

The `lastPoolCollection` mechanism has been completely removed and migrations to new pools now require the caller to pass an explicit pool collection to the `migratePool` function.

6.15 Undocumented Behavior

Design | Low | Version 1 | Code Corrected

The trade functions in `BancorNetwork` allow the user to declare a beneficiary. If the user sets this beneficiary to the 0-address, the beneficiary is replaced with the user's address. This behavior is not documented.

Code corrected:

All trade functions now explain the mentioned special behavior.

6.16 Unused Imports / Variables

Design | Low | Version 1 | Code Corrected

The following types have been imported but not used inside the respective contracts:

- `BancorNetwork`:
  - `WithdrawalRequest`
- `BancorV1Migration`:
  - `Upgradeable`
- `BNTPool`:
  - `Utils`
  - `Fraction`
  - `IPoolCollection`
  - `Pool`
  - `PoolToken`
- `PoolMigrator`:
  - `Fraction`
- `AutoCompoundingRewards`:
  - `AccessDenied`
- `StandardRewards`
  - `AccessDenied`

Additionally, `BNTPool` imports `Token` twice, `PoolMigrator` defines a private constant `INVALID_POOL_COLLECTION` that is not used and `StandardRewards` defines an unused error `PoolMismatch`.

Code corrected:

All unused imports and variables have been removed.

6.17 Wrong Function Name in `BancorPortal`

Design | Low | Version 1 | Code Corrected

`BancorPortal` contains a function `migrateSushiSwapV1Position` indicating calls to SushiSwap v1 even though the referenced contracts belong to SushiSwap v2.

Code corrected:

V1 and V2 strings have been removed from all function, event and variable names related to SushiSwap.

6.18 Wrong Interface

Correctness Low Version 1 Code Corrected

`BancorV1Migration.migratePoolTokens` types legacy `DSToken` addresses with v3's `IPoolToken` interface. While this is not a problem right now, future changes in the interface might create problems here.

Code corrected:

`BancorV1Migration` now uses a separate interface for legacy pool tokens.

6.19 `AutoCompoundingRewards` Can Burn More Pool Tokens Than Expected

Correctness Low Version 1 Code Corrected

On creation of a program in `AutoCompoundingRewards`, the caller provides the amount of tokens that should be distributed during the lifetime of the program. Depending on the token to be distributed, `createProgram` either calls `BNTPool.poolTokenAmountToBurn` and `PoolCollection.poolTokenAmountToBurn`. Both functions use the same formula to calculate the amount of pool tokens that have to be burned in order to distribute the given token amount:

```
function poolTokenAmountToBurn(uint256 bntAmountToDistribute) external view returns (uint256) {
    if (bntAmountToDistribute == 0) {
        return 0;
     }

     uint256 poolTokenSupply = _poolToken.totalSupply();
     uint256 val = bntAmountToDistribute * poolTokenSupply;

     return
         MathEx.mulDivF(
             val,
             poolTokenSupply,
             val + _stakedBalance * (poolTokenSupply - _poolToken.balanceOf(address(this)))
         );
}
```

The formula allows for the burning of high amounts of pool tokens (up to the total), which can become problematic for new deposits as the value of the pool tokens now far exceeds the value of the underlying tokens, potentially leading to large rounding errors for token suppliers. To highlight this problem, consider the following example:

- bnBNT total supply is 20000
- The protocol holds all bnBNT (i.e. no user has supplied any BNT)

In this case

```
_stakedBalance * (poolTokenSupply - _poolToken.balanceOf(address(this))
```

is now 0. The formula is therefore reduced to:

```
bntAmountToDistribute * poolTokenSupply * poolTokenSupply
    / bntAmountToDistribute * poolTokenSupply
```

The amount of tokens to be distributed is now completely factored out of the equation. It will therefore always return `poolTokenSupply` to be burned, no matter the amount of tokens to distribute.

Code corrected:

If the amount of pool tokens to be burnt in a single program exceeds 50% of the total supply, the program is now terminated. This ensures that the value of pool tokens does not appreciate too much in comparison to the underlying.

6.20 `BNTPool.renounceFunding` Fails on Insufficient BNT Pool Token Balance

Correctness | Low | Version 1 | Code Corrected

`BNTPool.renounceFunding` removes a given amount of BNT and burns the corresponding pool tokens. Because the pool tokens will be distributed to BNT liquidity providers, there can be circumstances where the protocol does not hold enough pool tokens for a given amount BNT that has to be burned. This will result in reverting transactions.

Consider the following example:

Liquidity provider withdrawals that exceed the amount of excess tokens in a given pool require the protocol to decrease the liquidity of the pool. If the amount of BNT liquidity that must be removed is greater than the amount of BNT pool tokens the protocol holds (because enough users have provided BNT liquidity in exchange for BNT pool tokens), the call will revert.

Code corrected:

`renounceFunding` now burns at most the amount of pool tokens available and updates the staked balance accordingly.

6.21 `ERC20Permit` Handling

Design | Low | Version 1 | Code Corrected

The `permit` function of `ERC20Permit` tokens is called expecting them to revert if the given signature is incorrect. While this is the correct behavior according to the `EIP 2612` specification, numerous token projects have shown that specifications are not always adhered to completely (e.g. the `transfer` function in `USDT`). Therefore, it might be possible that some token project exists that does not revert but rather returns a boolean value on calls to `permit`.

Code corrected:

``ERC20Permit` support has been completely removed from the project.

6.22 `MathEx.reducedFraction` Can Turn Denominator to 0

Correctness | Low | Version 1 | Code Corrected

`MathEx.reducedFraction` equally scales down two `uint256` values so that the higher value does not exceed a defined maximum. It computes the factor by which the values have to be divided with the following code:

```
uint256 scale = Math.ceilDiv(Math.max(fraction.n, fraction.d), max);
```

In the case that `fraction.d` is smaller than `scale`, the fraction's denominator will be set to 0 causing undefined behavior. Since the function is always used with a `max` value of `type(uint112).max`, this can only happen in edge cases where the numerator of the fraction is `type(uint112).max` times greater than the denominator.

Code Corrected:

`reducedFraction` now reverts if the denominator is set to 0.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 32 Bit Timestamps in Storage

Note Version 1

Some contracts (e.g. `AutoCompoundingRewards`) keep timestamps with the type `uint32` in storage. This will render the contracts unusable and make them hard to upgrade after the year 2106.

7.2 Impermanent Loss Protection Can Be Disabled

Note Version 2

`PoolCollection.enableProtection` can be called by Bancor v3's governance to disable Impermanent Loss protection. This can result in liquidity providers not being able to withdraw the full amount of tokens they are owed. In fact, LPs can end up with less tokens than originally provided and without any compensation.

7.3 Implementations Not Initialized

Note Version 1

Bancor deploys some if its contracts using a proxy pattern. However, the deployed implementations are not initialized by default. This is also evident on the current live version of the protocol. While this is not a problem currently, later changes might introduce `DELEGATECALL` op-codes. In this case, a malicious user could claim ownership of the contract and generate a `DELEGATECALL` to a contract containing a `SELFDESTRUCT` op-code, causing a denial of service.

7.4 Inconsistent Interface

Note Version 1

`PoolCollection` defines a function `enableDepositing` with a boolean argument to determine if depositing should be enabled or disabled. On the contrary, it defines the distinct functions `enableTrading` and `disableTrading`.

7.5 Liquidity Growth Not Restricted

Note Version 1

When a pool gets enabled for trading, the starting liquidity is set to a pre-determined amount of BNT and not set to the full amount possible. On certain actions, the trading liquidity is allowed to grow by the `LIQUIDITY_GROWTH_FACTOR` factor. However, anyone can grow the liquidity by calling

BancorNetwork.depositFor with the minimum amount of 1 wei token. Thus, this mechanism only protects against accidents and not against deliberate manipulation.

7.6 Missing Access Control in postUpgrade

Note **Version 1**

postUpgrade has no access restrictions. Using it in an upgrade needs to happen in one transaction if frontrunning should be mitigated. Bancor has a deploy script that will automatically call this function in the upgrade transaction. But this is not guaranteed and might fail. We cannot see a case that it should be callable by everyone.

7.7 More vBNt Than bnBNT Obtainable

Note **Version 1**

If the protocol does not have any bnBNT, it should be impossible to deposit BNT and hence obtain vBNT. However, in this case, a user can first send some bnBNT to the BNTPool and then deposit some BNT to obtain both the pre-owned bnBNT and newly minted vBNT. Although the BNT the user just deposited is not withdrawable anymore as he does no longer own the corresponding bnBNT, he is able to obtain additional vBNT at this cost.

7.8 No Recovery of Accidental Token Transfers Possible

Note **Version 1**

In case an ERC20 token other than the BNT or one of the base tokens is sent to the contract, then it cannot be recovered. Among other reasons, this might happen due to airdrops based on the base tokens.

7.9 Potential External Contract Manipulation

Note **Version 1**

The functions PoolCollection.withdrawalAmounts and BancorNetworkInfo.withdrawalAmounts are subject to manipulation by reentrancy. If the mentioned functions are used in any way to alter the state of an external contract (for example an investment protocol that supplies liquidity to Bancor v3), the values they return can be manipulated by calling the external contract in the onFlashLoan callback of BancorNetwork.flashloan.

This is possible because of the following call in PoolCollection._poolWithdrawalAmounts:

```
int256 baseTokenExcessAmount = pool.balanceOf(address(_masterVault)) -
    data.liquidity.baseTokenTradingLiquidity;
```

onFlashLoan will be called after the balance of _masterVault has already been reduced by the flashloan amount.

7.10 Redundant Role Management

Note **Version 1**

Most contracts are upgradable. Hence, they have their own admin account. There is no central management checking these roles are set accordingly. An admin change needs to be done individually and redundantly.

7.11 Unequal Token Burning

Note **Version 2**

Withdrawal of supplied tokens is subject to a 7-day waiting period. In this period, newly generated interest is not accrued to the withdrawing user. This means, that after 7-days, the pool tokens sent to the `PendingWithdrawals` contract are worth slightly more than the amount of tokens the user actually receives. Because `PoolCollection` completely burns all of these pool tokens, while `BNTPool` keeps them, the outcome of a withdrawal of BNT differs to withdrawals of other tokens. Consider the following 2 examples:

BNT:

- `totalSupply` of bnBNT is 100.
- `_stakedBalance` in the `BNTPool` is 100.
- A user initiates a withdrawal with 50 bnBNT, allowing them to withdraw 50 BNT after 7 days.
- After 7 days, 100% interest has accrued and the new `_stakedBalance` is now 200.
- The user withdraws their 50 BNT (which get minted) and the 50 bnBNT are repossessed by `BNTPool`.
- `_stakedBalance` is now 200.
- `totalSupply` is now 100.

TKN:

- `totalSupply` of bnTKN is 100.
- `_stakedBalance` in the `PoolCollection` is 100.
- A user initiates a withdrawal with 50 bnTKN, allowing them to withdraw 50 TKN after 7 days.
- After 7 days, 100% interest has accrued and the new `_stakedBalance` is now 200.
- The user withdraws their 50 TKN and the 50 bnTKN are burned.
- `_stakedBalance` is now 150.
- `totalSupply` is now 50.

Both examples illustrate the same scenario but in the BNT case, a pool token is worth 2 BNT in the end, while in the TKN case, a pool token is now worth 3 TKN.

7.12 Unsupported Tokens

Note **Version 1**

Not all `ERC20` tokens can act as base tokens for Bancor v3 contracts. In particular, the following tokens are **not** supported:

- Tokens that return metadata fields like `name` and `symbol` encoded as `bytes32` instead of `string` (e.g. `MKR`). `PoolTokenFactory.createPoolToken` will fail to create a pool token for these tokens.
- Tokens that take a fee on transfer (e.g. `PAXG` and possibly `USDT`). A `deposit` will use the full amount to mint pool tokens while the contract has received a lower amount.
- Tokens that have a rebasing mechanism (e.g. AAVE's `aToken`). User's staked balances will not be updated accordingly.

Additionally, the following tokens could break the protocol in the future:

- Tokens with blacklists (e.g. `USDT`, `USDC`).
- Upgradeable tokens that add one of the mentioned mechanisms in the future.
- Pausable tokens (e.g. `BNB`).