

# Bancor V2 AMM Security Audit

- [1 Executive Summary](#)
- [2 Scope](#)
  - [2.1 Objectives](#)
  - [2.2 Key Observations](#)
- [3 Recommendations](#)
- [4 System Overview](#)
  - [4.1 Actors](#)
- [5 Issues](#)
  - [5.1 Oracle updates can be manipulated to perform atomic front-running attack](#) **Critical**  
✓ Addressed
  - [5.2 Slippage and fees can be manipulated by a trader](#) **Major** ✓ Addressed
  - [5.3 Loss of the liquidity pool is not equally distributed](#) **Major** ✓ Addressed
  - [5.4 Oracle front-running could deplete reserves over time](#) **Major** ✓ Addressed
  - [5.5 Use of external calls with a fixed amount of gas](#) **Medium** Won't Fix
  - [5.6 Use of assert statement for input validation](#) **Minor** ✓ Addressed
- [6 Bytecode Verification](#)
  - [6.1 Results](#)
- [Appendix 1 - Code Quality Recommendations](#)
  - [A.1.1 Increase test coverage](#)
- [Appendix 2 - Files in Scope](#)
  - [A.2.1 Contracts Description Table](#)
  - [A.2.2 Legend](#)
- [Appendix 3 - Disclosure](#)

**Date**

June 2020

# 1 Executive Summary

---

This report presents the results of our engagement with Bancor to review version 2 of the Bancor Liquidity pool. The initial review was conducted over the course of two weeks, from July 15th to July 26th, 2020 by Bernhard Mueller and Sergii Kravchenko. A total of 15 person-days were spent.

During the first week, the auditors familiarized themselves with the existing Bancor codebase as well as the novel concepts introduced with version 2 of the liquidity pool. They assessed the code quality and checked for basic vulnerabilities, such as integer overflow bugs and improper access permissions, and performed a preliminary analysis with regards to the business logic. During the second week the auditors focused on the behavior of the new liquidity pool model and its incentive mechanisms.

Several issues were discovered which were subsequently addressed by the Bancor team. These fixes were reviewed over the course of two days, July 28th and July 29th (a total of 4 person-days).

## 2 Scope

---

The initial 2-week review was conducted on a private Github repository while the codebase was still under development. The list of files covered in the audit can be found in the [Appendix](#).

The review of the fixes occurred on a [recent commit](#) on the public Bancor contracts repository.

The total time budget of 3 person-weeks allotted to the main review phase was limited considering the complexity of the system. Additionally, the code underwent several changes throughout the audit. The review was performed on a best-effort basis with a focus on covering as much ground as possible given the time available.

The mitigations provided by the Bancor team were reviewed over the course of 4 person-days, which was sufficient to confirm that the mitigations are somewhat effective without exhaustively assessing their overall impact on the system. Due to its high complexity, the system needs to be tested extensively under real-world conditions.

### 2.1 Objectives

Together with the the Bancor team, we identified the following priorities for our review:

1. Identify known vulnerabilities particular to smart contract systems, as outlined in our [Smart Contract Best Practices](#), and the [Smart Contract Weakness Classification Registry](#);
2. Review the novel AMM mechanisms as well as the correctness of the code implementing the new formulas;
3. Evaluate whether the system is consistently implements the intended functionality and identify ways of gaming the system.

### 2.2 Key Observations

The code is generally of good quality and well-readable. The new formulas introduced with v2 come with extensive test coverage and have been shown to behave correctly under regular market conditions. That said, given the complexity of the system it is difficult to assess whether the results incentives and fee mechanisms will produce the desired results in a real-world scenario.

In the initial phase of the audit, the auditors demonstrated several ways of extracting value from the pool. For instance, arbitrageurs could frontrun Oracle updates and make conversions that took the future rebalancing into account (6.4). In an early version, this arbitrage could be performed atomically by trading at a stale rate, triggering the Oracle update and swapping in the same transaction (6.1). Users could also minimize slippage and fees by atomically adding and removing large amounts of liquidity (e.g. provisioned via a flash loan) before and after a conversion (see 6.2 and 6.3).

Several mitigations for these issues were put in place in the final version that make these attacks more difficult and to perform and riskier for the attackers. However, the mitigations also increased the complexity of the system.

Fundamentally, the weight rebalancing mechanism creates an incentive to align the balance of the primary reserve (i.e., the reserve containing an arbitrary token we'll refer to as "XYZ") such that deficits in that reserve are compensated at the cost of liquidity providers (LPs) who stake in the secondary

reserve (which usually contains BNT). A dynamic fee calculation mechanism has been added in the mitigation phase to counterbalance the risk of BNT reserve balance permanently falling below the staked balance. This mechanism charges users additional swap fees whenever the actual BNT balance is in deficit (assuming the XYZ reserve is balanced). These additional fees are directed to decrease the deficit. Ultimately however, some risk remains that the pool contains insufficient reserves and not all LPs can withdraw their stake. This risk primarily affects BNT LPs, but XYZ reserves may also have insufficient funds if the total reserves deficit is large enough.

LPs should be adequately informed about the financial risks associated with providing liquidity to either reserve. It is also recommended to carefully monitor the system in production with limited liquidity.

### 3 Recommendations

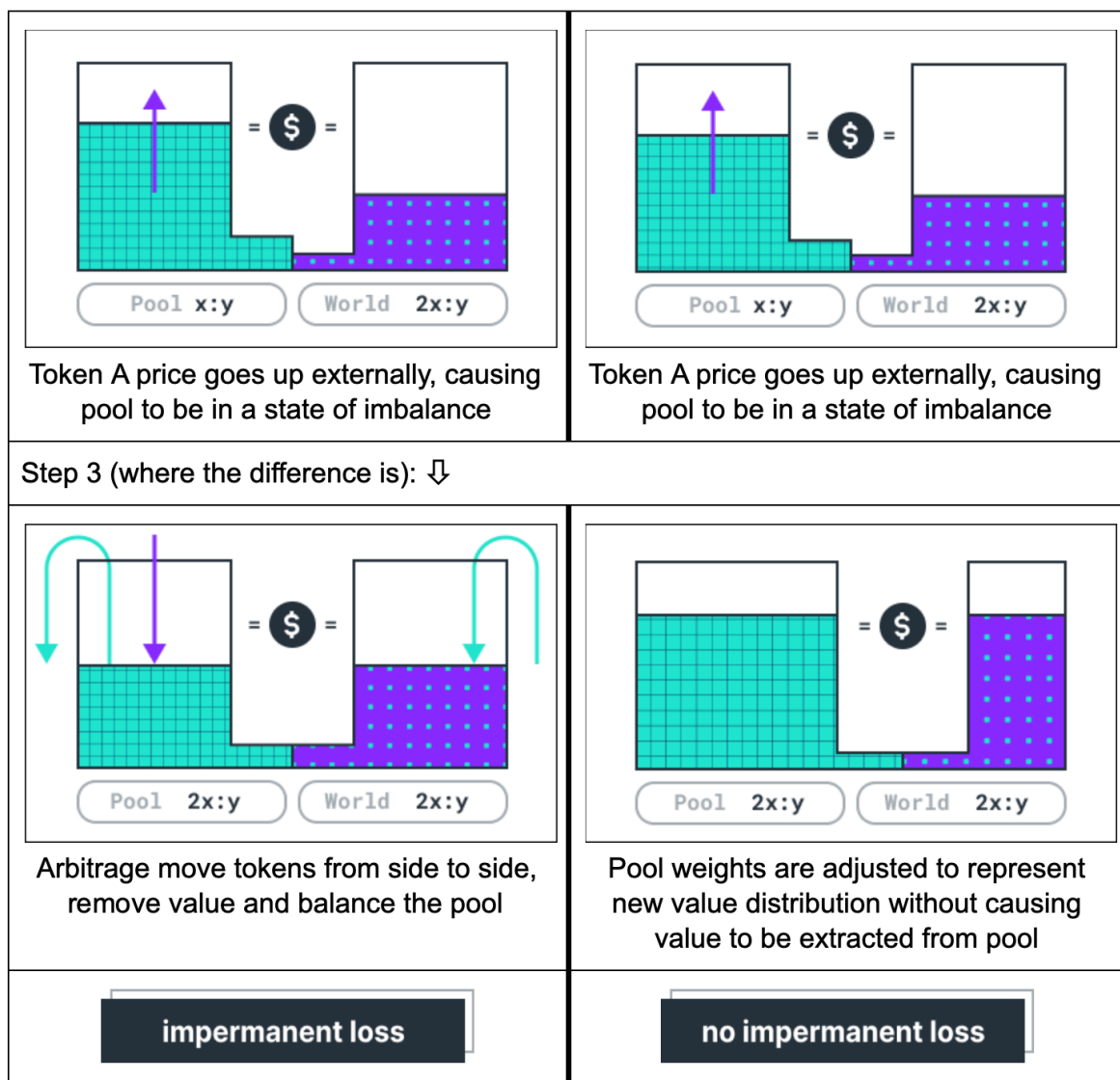
---

- Carefully review the issues and code quality recommendations described in this report.
- Create an exhaustive threat model describing all edge cases and possible attack vectors with respect to variants of frontrunning and/or gaming the incentive structure created by the Bancor V2 formula. Some potential attacks have been outlined in this report but the analysis is not exhaustive. Clearly inform liquidity providers about the risks associated with staking tokens in the pool.
- Once the code and formula is finalized, it would be ideal to perform a detailed audit of the overall system architecture in combination with a bug bounty program. Notably, Bancor is already planning to launch with a beta pilot that limits the maximum amount of liquidity per pool.

## 4 System Overview

Existing liquidity pools, including Bancor V1, require liquidity providers to add determinate amounts of each of the underlying assets to the pool. This exposes providers to a downside risk called *impermanent loss*: If the relative external market price of the assets diverges from the price when liquidity was provided, the value of the assets the provider can withdraw may be less than the combined value when the liquidity was added.

Bancor V2 seeks to eliminate the risk of impermanent loss by allowing users to provide liquidity with up to 100% exposure to a single ERC20 token. This is achieved by dynamically adjusting the relative weights of the reserves based on an external Oracle. With every “rebalancing”, the weights are adjusted such that the pool represents the external price ratio between the token in the primary reserve (which contains an arbitrary ERC20 token, referred to as XYZ in technical documentation) and the token in the secondary reserve (usually BNT).



An additional feature of V2 is “Virtual Amplification”, a multiplier that virtually inflates the staked reserves to allow for lower slippage in conversions. The idea is to allow for more and larger conversions to occur on the pool, thus generating more fees for liquidity providers.

## 4.1 Actors

The relevant actors are listed below with their respective abilities:

*Liquidity pool owners* can:

- Update & enable the conversion whitelist. When enabled, only addresses that are whitelisted are allowed to use the converter.
- Set the conversion fee (bound by the max conversion fee set during construction).
- Withdraw any ERC20 tokens held by the anchor. In v2, the anchor is an instance of `PoolTokensContainer` that contains the two pool tokens.
- In the case of an upgrade, the pool owner temporarily transfers ownership to the Bancor upgrader smart contract, which - once granted ownership - has the additional ability to withdraw tokens and Ether from the pool (in order to transfer it to a new contract account).

*Pool users* can:

- Add or remove liquidity to/from the pool, which results in pool tokens to be minted or burned. Note that while in v1 there is only a single pool token per pool, in v2 each pool is anchored to exactly two pool tokens (one per reserve).
- Convert between the tokens in reserve. Note that the `convert()` function is only callable via the Bancor Network contract which was not within the scope of this audit.

## 5 Issues

---

Each issue has an assigned severity:

- **Minor** issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- **Medium** issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- **Major** issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- **Critical** issues are directly exploitable security vulnerabilities that need to be fixed.

### 5.1 Oracle updates can be manipulated to perform atomic front-running attack **Critical** ✓ Addressed

#### Resolution

The issue was mitigated by updating the Oracle price only once per block and consistently only using the old value throughout the block instead of querying the Oracle when adding or removing liquidity. Arbitrageurs can now no longer do the profitable trade within a single transaction which also precludes the possibility of using flash loans to amplify the attack.

#### Description

It is possible to atomically arbitrage rate changes in a risk-free way by “sandwiching” the Oracle update between two transactions. The attacker would send the following 2 transactions at the moment the Oracle update appears in the mempool:

The first transaction, which is sent with a higher gas price than the Oracle update transaction, converts a very small amount. This “locks in” the conversion weights for the block since `handleExternalRateChange()` only updates weights once per block. By doing this, the arbitrageur ensures that the stale Oracle price is initially used when doing the first conversion in the following transaction.

The second transaction, which is sent at a slightly lower gas price than the transaction that updates the Oracle, does the following:

1. Perform a large conversion at the old weight;
2. Add a small amount of Liquidity to trigger rebalancing;
3. Convert back at the new rate.
4. The attacker can also leverage the incentive generated by the formula by converting such that  $\text{primary reserve balance} == \text{primary reserve staked balance}$ .



The attacker can obtain liquidity for step 2 using a flash loan. The attack will deplete the reserves of the pool. An example is shown in section 5.4.

### **Recommendation**

Do not allow users to trade at a stale Oracle rate and trigger an Oracle price update in the same transaction.

## **5.2 Slippage and fees can be manipulated by a trader** Major

✓ Addressed

Resolution

The issue was addressed by introducing an exit fee mechanism. When a liquidity provider wants to withdraw some liquidity, the smart contract returns fewer tokens if the primary reserve is not in the balanced state. So in most cases, the manipulations described in the issue should potentially be non-profitable anymore. Although, in some cases, the traders still may have some incentive to add liquidity before making the trade and remove it after to get a part of the fees (i.e., if the pool is going to be in a balanced state after the trade).

## Description

Users are making trades against the liquidity pool (converter) with slippage and fees defined in the converter contract and Bancor formula. The following steps can be done to optimize trading costs:

- Instead of just making a trade, a user can add a lot of liquidity (of both tokens, or only one of them) to the pool after taking a flash loan, for example.
- Make the trade.
- Remove the added liquidity.

Because the liquidity is increased on the first step, slippage is getting smaller for this trade. Additionally, the trader receives a part of the fees for this trade by providing liquidity.

One of the reasons why this is possible is described in another issue [issue 5.3](#).

This technique of reducing slippage could be used by the trader to get more profit from any frontrunning/arbitrage opportunity and can help to deplete the reserves.

## Example

Consider the initial state with an amplification factor of 20 and zero fees:

```
Initial state:  
converter TKN balance = 10000000  
converter TKN weight = 500000  
converter BNT balance = 10000000  
converter BNT weight = 500000
```

Here a user can make a trade with the following rate:

```
-> Convert 9000000 TKN into 8612440 BNT.
```

But if the user adds 100% of the liquidity in both tokens before the trade, the slippage will be lower:

```
-> Convert 9000000 TKN into 8801955 BNT.
```

## Recommendation

Fixing this issue requires some modification of the algorithm.

## 5.3 Loss of the liquidity pool is not equally distributed **Major**

✓ Addressed

### Resolution

The issue was addressed by adding a new fee mechanism called 'adjusted fees'. This mechanism aims to decrease the deficit of the reserves over time. If there is a deficit of reserves, it is usually present on the secondary token side, because there is a strong incentive to bring the primary token to the balanced state. Roughly speaking, the idea is that if the secondary token has a deficit in reserves, there are additional fees for trading that token. These fees are not distributed across the liquidity providers like the regular fees. Instead, they are just populating the reserve, decreasing the existing deficit.

Loss is still not distributed across the liquidity providers, and there is a possibility that there are not enough funds for everyone to withdraw them. In the case of a run on reserves, LPs will be able to withdraw funds on a first-come-first-serve basis.

### Description

All stakeholders in the liquidity pool should be able to withdraw the same amount as they staked plus a share of fees that the converter earned during their staking period.

**code/contracts/converter/LiquidityPoolV2Converter.sol:L491-L505**

```
IPoolTokensContainer(anchor).burn(_poolToken, msg.sender, _amount);

// calculate how much liquidity to remove
// if the entire supply is liquidated, the entire staked amount should be
sent, otherwise
// the price is based on the ratio between the pool token supply and the
staked balance
uint256 reserveAmount = 0;
if (_amount == initialPoolSupply)
    reserveAmount = balance;
else
    reserveAmount = _amount.mul(balance).div(initialPoolSupply);

// sync the reserve balance / staked balance
reserves[reserveToken].balance =
reserves[reserveToken].balance.sub(reserveAmount);
uint256 newStakedBalance = stakedBalances[reserveToken].sub(reserveAmount);
stakedBalances[reserveToken] = newStakedBalance;
```

The problem is that sometimes there might not be enough funds in reserve (for example, due to this [issue 5.4](#)). So the first ones who withdraw their stakes receive all the tokens they own. But the last stakeholders might not be able to get their funds back because the pool is empty already.

So under some circumstances, there is a chance that users can lose all of their staked funds.

This issue also has the opposite side: if the liquidity pool makes an extra profit, the stakers do not owe this profit and cannot withdraw it.

### **Recommendation**

Distribute losses evenly across the liquidity providers.

## 5.4 Oracle front-running could deplete reserves over time Major

✓ Addressed

### Resolution

To mitigate this issue, the Bancor team has added a mechanism that adjusts the effective weights once per block based on its internal price feed. The conversion rate re-anchors to the external oracle price once the next oracle update comes in. This mechanism should help to cause the weight rebalancing caused by the external Oracle update to be less pronounced, thereby limiting the profitability of Oracle frontrunning. It should be noted that it also adds another layer of complexity to the system. It is difficult to predict the actual effectiveness and impact of this mitigation measure without simulating the system under real-world conditions.

### Description

Bancor's weight rebalancing mechanism uses Chainlink price oracles to dynamically update the weights of the assets in the pool to track the market price. Due to Oracle price updates being visible in the mempool before they are included in a block, it is always possible to know about Oracle updates in advance and attempt to make a favourable conversion which takes the future rebalancing into account, followed by the reverse conversion after the rebalancing has occurred. This can be done with high liquidity and medium risk since transaction ordering on the Ethereum blockchain is largely predictable.

Over time, this could deplete the secondary reserve as the formula compensates by rebalancing the weights such that the secondary token is sold slightly below its market rate (this is done to create an incentive to bring the primary reserve back to the amount staked by liquidity providers).

### Example

Consider the initial state with an amplification factor of 20 and zero fees:

```
converter TKN balance = 100,000,000
converter TKN weight = 500,000
converter BNT balance = 100,000,000
converter BNT weight = 500,000
frontrunner TKN balance = 100,000,000
frontrunner BNT balance = 0
```

```
Oracle A rate = 10,000
Oracle B rate = 10,000
```

The frontrunner sees a Chainlink transaction in the mempool that changes Oracle B rate to 10,500. He sends a transaction with a slightly higher gas price than the Oracle update.

- Convert 1,000,000 TKN into 999,500 BNT.

The intermediate state:

```
converter TKN balance = 101,000,000
converter TKN weight = 500,000
converter BNT balance = 99,000,500
converter BNT weight = 500,000
frontrunner TKN balance = 99,000,000
frontrunner BNT balance = 999,500
```

In the following block, the frontrunner sends another transaction with a high gas price (the goal is to be first to convert at the new rate set by the Oracle update):

- Convert 999,500 BNT back into TKN.

The state is:

```
converter TKN balance = 99,995,006
converter TKN weight = 498,754
converter BNT balance = 100,000,000
converter BNT weight = 501,246
frontrunner TKN balance = 100,004,994
frontrunner BNT balance = 0
```

The frontrunner can now leverage the incentive created by the formula to bring back TKN reserve balance to staked TKN balance by converting TKN back to BNT:

- Convert 4,994 TKN to BNT

The final state is:

```
converter TKN balance = 100,000,000
converter TKN weight = 498,754
converter BNT balance = 99,995,031
converter BNT weight = 501,246
frontrunner TKN balance = 100,000,000
frontrunner BNT balance = 4,969
```

The pool is now balanced and the frontrunner has gained 4,969 BNT.

## Recommendation

This appears to be a fundamental problem caused by the fact that rebalancing is predictable. It is difficult to assess the actual impact of this issue without also reviewing components external to the scope of this audit (Chainlink) and extensively testing the system under real-world conditions.

**5.5 Use of external calls with a fixed amount of gas** **Medium** **Won't Fix**

**Resolution**

It was decided to accept this minor risk as the usage of `.call()` might introduce other unexpected behavior.

## Description

The converter smart contract uses the Solidity `transfer()` function to transfer Ether.

`.transfer()` and `.send()` forward exactly 2,300 gas to the recipient. The goal of this hardcoded gas stipend was to prevent reentrancy vulnerabilities, but this only makes sense under the assumption that gas costs are constant. Recently EIP 1884 was included in the Istanbul hard fork. One of the changes included in EIP 1884 is an increase to the gas cost of the SLOAD operation, causing a contract's fallback function to cost more than 2300 gas.

## Examples

### `code/contracts/converter/ConverterBase.sol:L228`

```
_to.transfer(address(this).balance);
```

### `code/contracts/converter/LiquidityPoolV2Converter.sol:L370`

```
if (_targetToken == ETH_RESERVE_ADDRESS)
```

### `code/contracts/converter/LiquidityPoolV2Converter.sol:L509`

```
msg.sender.transfer(reserveAmount);
```

## Recommendation

It's recommended to stop using `.transfer()` and `.send()` and instead use `.call()`. Note that `.call()` does nothing to mitigate reentrancy attacks, so other precautions must be taken. To prevent reentrancy attacks, it is recommended that you use the checks-effects-interactions pattern.

## 5.6 Use of assert statement for input validation Minor ✓ Addressed

### Resolution

Assertions are no longer used in the final version reviewed.



## Description

Solidity assertion should only be used to assert invariants, i.e. statements that are expected to always hold if the code behaves correctly. Note that all available gas is consumed when an assert-style exception occurs.

## Examples

It appears that `assert()` is used in one location within the test scope to catch invalid user inputs:

**code/contracts/converter/LiquidityPoolV2Converter.sol:L354**

```
assert(amount < targetReserveBalance);
```

## Recommendation

Using `require()` instead of `assert()`.

## 6 Bytecode Verification

---

Bytecode-level checking helps to ensure that the code behaves correctly for all input values. In this audit we used [Mythx](#) deep analysis to verify a small number of basic properties on the weight rebalancing and conversion functions and to detect conditions that would cause runtime exceptions. MythX uses symbolic execution and input fuzzing to explore a large amount of possible inputs and program states.

Note that the Bancor formula is compiled with solc-0.4.25 / 20,000 optimization passes.

We checked whether the following properties hold for all inputs:

- [P1] Function `balancedWeights`: Sum of weights returned by must equal `MAX_WEIGHT`
- [P2a] Function `crossReserveTargetAmount`: Output amount must not be greater than target reserve balance
- [P2b] Function `crossReserveTargetAmount`: If reserve balances are equal and source weight < target weight, target amount must be lower than input amount

Note that `balancedWeights` is known to revert when  $(t * p) / (r * q) * \log(s / t)$  is not in the range  $[-1/e, 1/e]$ , where:

- `t` is the primary reserve staked balance
- `s` is the primary reserve current balance
- `r` is the secondary reserve current balance
- `q` is the primary reserve rate
- `p` is the secondary reserve rate

The following preconditions were set on the input to reflect realistic input ranges. For `balancedWeights`:

```
require(_primaryReserveStakedBalance > 0);
require(_primaryReserveBalance > 0);
require(_secondaryReserveBalance > 0);
require(_reserveRateNumerator > 0);
require(_reserveRateDenominator > 0);
require(_reserveRateNumerator < 10 ** 6);
require(_reserveRateDenominator < 10 ** 6);
require(_primaryReserveStakedBalance <= 10**30);
require(_primaryReserveBalance <= 10**30);
require(_secondaryReserveBalance <= 10**30);
```

For `crossReserveTargetAmount`:

```
require(_sourceReserveBalance > 0);
require(_targetReserveBalance > 0);
require(_sourceReserveBalance <= 10**30);
require(_targetReserveBalance <= 10**30);
require(_sourceReserveWeight + _targetReserveWeight == MAX_WEIGHT);
```

```
require(_amount > 0);  
require(_amount <= 10**30);
```

## **6.1 Results**

No violations of the properties tested were found. Our tools also did not identify any cases that would cause the function to revert for the given input ranges.

# Appendix 1 - Code Quality Recommendations

## A.1.1 Increase test coverage

While test coverage for BancorFormula.sol is high, the tests for ConverterBase.sol and LiquidityPoolV2Converter.sol only reach ~67% of statements and less than 50% of branches. In general we recommend aiming for near-100% test coverage.


File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/converter/	66.67	42.86	47.21	58.94	
BancorFormula.sol	94.03	67.65	77.78	90.41	... 4,1147,1159
ConverterBase.sol	69.01	41.67	75	71.05	... 406,432,548
LiquidityPoolV2Converter.sol	66.67	47.92	51.72	66.27	... 548,549,601
PoolTokensContainer.sol	93.33	50	83.33	93.33	86
contracts/utility/	60	39.29	70.83	61.6	
Owned.sol	100	66.67	100	100	
PriceOracle.sol	61.9	16.67	66.67	63.64	... 100,114,115
ReentrancyGuard.sol	100	50	100	100	
TokenHandler.sol	100	50	100	100	
TokenHolder.sol	0	100	0	0	35
Utils.sol	100	66.67	100	100	
Whitelist.sol	0	0	0	0	... 74,75,84,85
-----	-----	-----	-----	-----	-----

## Appendix 2 - Files in Scope

This audit focused on the files related to `LiquidityPoolV2Converter` as well as newly added functions in `BancorFormula`.

File	SHA-1 hash
contracts/converter/BancorFormula.sol	097b6424e61614a1b50751287d7c53361816bc
contracts/converter/ConverterBase.sol	289a5d5eb28f25bd5ca44874702d2b434633c
contracts/converter/LiquidityPoolV2Converter.sol	f321b59217451179cf82fe6a3cadd07c5c941578z
contracts/converter/PoolTokensContainer.sol	813b0a091100b56511fe5f78346d99f74aee8ec
contracts/utility/TokenHandler.sol	f0f6f6f2e62bb529270af8c66f7d202ff22f849z
contracts/utility/TokenHolder.sol	91292c475bd34ca893f428e811d366c1b07a0z
contracts/utility/Owned.sol	b9732bd40652fac0cbc8d06b02a32b42bdefa
contracts/utility/PriceOracle.sol	2coda6b8fe40f42639f29e436b4ef312ef62c6z
contracts/utility/ReentrancyGuard.sol	dc59150282a9058a974afeb39f7a59ed4cfe3ez



### A.2.1 Contracts Description Table

Contract	Type	Bases
L	Function Name	Visibility
<b>LiquidityPoolV2Converter</b>	Implementation	LiquidityPoolConverter
L		Public !
L	<code>_validPoolToken</code>	Internal 
L	<code>converterType</code>	Public !
L	<code>isActive</code>	Public !
L	<code>activate</code>	Public !
L	<code>reserveStakedBalance</code>	Public !
L	<code>setReserveStakedBalance</code>	Public !
L	<code>setMaxStakedBalances</code>	Public !

<b>Contract</b>	<b>Type</b>	<b>Bases</b>
L	disableMaxStakedBalances	Public !
L	poolToken	Public !
L	liquidationLimit	Public !
L	addReserve	Public !
L	targetAmountAndFee	Public !
L	doConvert	Internal 🔒
L	addLiquidity	Public !
L	removeLiquidity	Public !
L	adjustedFee	Internal 🔒
L	targetAmountAndFee	Private 🔒
L	handleExternalRateChange	Private 🔒
L	rebalance	Public !
L	newReserveWeights	Private 🔒
L	dispatchRateEvents	Private 🔒
L	dispatchTokenRateUpdateEvent	Private 🔒
L	dispatchPoolTokenRateUpdateEvent	Private 🔒
<b>BancorFormula</b>	Implementation	IBancorFormula
L	initMaxExpArray	Private 🔒
L	initLambertArray	Private 🔒
L	init	Public !
L	purchaseTargetAmount	Public !
L	saleTargetAmount	Public !
L	crossReserveTargetAmount	Public !
L	fundCost	Public !
L	liquidateReserveAmount	Public !

<b>Contract</b>	<b>Type</b>	<b>Bases</b>
L	balancedWeights	Public !
L	power	Internal 🔒
L	generalLog	Internal 🔒
L	floorLog2	Internal 🔒
L	findPositionInMaxExpArray	Internal 🔒
L	generalExp	Internal 🔒
L	optimalLog	Internal 🔒
L	optimalExp	Internal 🔒
L	lowerStake	Internal 🔒
L	higherStake	Internal 🔒
L	lambertPos1	Internal 🔒
L	lambertPos2	Internal 🔒
L	lambertPos3	Internal 🔒
L	lambertNeg1	Internal 🔒
L	balancedWeightsByStake	Internal 🔒
L	normalizedWeights	Internal 🔒
L	accurateWeights	Internal 🔒
L	roundDiv	Internal 🔒
L	calculatePurchaseReturn	Public !
L	calculateSaleReturn	Public !
L	calculateCrossReserveReturn	Public !
L	calculateCrossConnectorReturn	Public !
L	calculateFundCost	Public !
L	calculateLiquidateReturn	Public !
L	purchaseRate	Public !
L	saleRate	Public !
L	crossReserveRate	Public !
L	liquidateRate	Public !

## A.2.2 Legend

Symbol	Meaning
	Function can modify state
	Function is payable



## Appendix 3 - Disclosure

---

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

**PURPOSE OF REPORTS** The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

**LINKS TO OTHER WEB SITES FROM THIS WEB SITE** You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

**TIMELINESS OF CONTENT** The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

