SMART CONTRACT AUDIT REPORT

for

Bancor V3

Prepared By: Patrick Lou

PeckShield

April 18, 2022

Document Properties

Client	Bancor
Title	Smart Contract Audit Report
Target	Bancor V3
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Shulin Bie, Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 18, 2022	Xuxian Jiang	Final Release
1.0-rc	April 11, 2022	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Bancor V3` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Bancor

The `Bancor` protocol is a fully on-chain liquidity protocol that can be implemented on any smart contract-enabled blockchain. It pioneers the new way of `AMM`-based trading that allows for buying and selling tokens against a smart contract. The audited `Bancor V3` introduces a new kind of composable single-sided pool token that only rises in relation to the staked asset, making them the ideal collateral and an excellent `DeFi` money lego. It also revises its tokenomics to enable a more cost-efficient system for `IL` protection and create greater deflationary pressure on `BNT`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Bancor V3

Item	Description
Issuer	Bancor
Website	http://bancor.network/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 18, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/bancorprotocol/contracts-v3.git (4d4bee9)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/bancorprotocol/contracts-v3.git (4f36eb4)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Bancor V3` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	5	■ ■ ■ ■ ■
Informational	1	■
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Denial-Of-Service in Pool Creation	Business Logic	Fixed
PVE-002	High	Incorrect targetBalance Calculation in Token Swaps	Business Logic	Fixed
PVE-003	Informatinoal	Improved Logic in Cancelling Pending Withdrawals	Coding Practices	Fixed
PVE-004	Low	Proper TotalLiquidityUpdated Event Generation	Business Logic	Fixed
PVE-005	Low	Improved Initialization Logic in AutoCompoundingStakingRewards	Coding Practices	Fixed
PVE-006	Low	Proper Removal Of _programByPool In terminateProgram()	Business Logic	Fixed
PVE-007	Low	Proper Enforcement of depositingEnabled in PoolCollection	Coding Practices	Fixed
PVE-008	Medium	Proper Fee Collection in BancorNetwork::_tradeBNT()	Business Logic	Fixed
PVE-009	Medium	Trust on Admin Keys	Security Features	Mitigated

Beside the identified issues, we note that the staking support assumes the staked tokens are not deflationary. Also, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Denial-Of-Service in Pool Creation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BancorNetwork`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Bancor V3` protocol whitelists the pool tokens that are supported for swaps and enables dynamic creation of the liquidity pools. While analyzing the logic behind the dynamic pool creation, we notice the current implementation may exhibit a denial-of-service situation that needs to be corrected.

To elaborate, we show below the affected `createPool()` function. This is a public function, but it can only be used to create a pool with a whitelisted pool token. By design, a pool is tagged with a `poolType` and, for the same pool token, it is not allowed to create multiple pools with different pool `poolType`. With that, it comes to our attention that the current implementation may disallow a pool for the intended `poolType` to be created.

```
    function createPool(uint16 poolType, Token token) external nonReentrant validAddress
        (address(token)) {
        if (_isBNT(token)) {
            revert InvalidToken();
        }

        if (!_liquidityPools.add(address(token))) {
            revert AlreadyExists();
        }

        // get the latest pool collection, corresponding to the requested type of the
            new pool, and use it to create the
        // pool
        IPoolCollection poolCollection = _latestPoolCollections[poolType];
        if (address(poolCollection) == address(0)) {
```

```
            revert InvalidType();
        }

        // this is where the magic happens...
        poolCollection.createPool(token);

        // add the pool collection to the reverse pool collection lookup
        _collectionByPool[token] = poolCollection;

        emit PoolAdded({ poolType: poolType, pool: token, poolCollection: poolCollection
            });
    }
```

Listing 3.1: `BancorNetwork::createPool()`

Specifically, a malicious actor may pre-create a pool but for a different `poolType`, which blocks the pool creation for the intended `poolType`!

Recommendation Revise the above logic to eliminate the possible denial-of-service situation.

Status This issue has been fixed in this commit: `4b75ec4`.

3.2 Incorrect targetBalance Calculation in Token Swaps

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `PoolCollection`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Bancor V3` protocol is in essence a `DEX` protocol that has the built-in support of swapping one token to another. And each swap may have the cost of being charged for the swap fee. While reviewing the current logic of fee collection, we notice the current implementation is flawed and should be corrected.

To elaborate, we show below the core swap routine (`_processTrade()`). For simplicity, we examine the swap case when `bySourceamount` and `isSourceBNT` are `true` and `false`, respectively. With that, this routine makes use of the `_tradeAmountAndFeeBySourceAmount()` helper to compute the `tradeAmountAndFee`, which contains the expected target token amount (in `result.targetAmount` – line 1377) after the conversion. It comes to our attention that this expected target token amount already removes the `network fee`. In other words, the update to the pool's target balance `result.targetBalance` (line 1043) does not take into account the `network fee`.

```
    function _processTrade(TradeIntermediateResult memory result) private view {
        TradeAmountAndTradingFee memory tradeAmountAndFee;

        if (result.bySourceAmount) {
            tradeAmountAndFee = _tradeAmountAndFeeBySourceAmount(
                result.sourceBalance,
                result.targetBalance,
                result.tradingFeePPM,
                result.sourceAmount
            );

            result.targetAmount = tradeAmountAndFee.amount;

            // ensure that the target amount is above the requested minimum return amount
            if (result.targetAmount < result.limit) {
                revert InsufficientTargetAmount();
            }
        } else {
            tradeAmountAndFee = _tradeAmountAndFeeByTargetAmount(
                result.sourceBalance,
                result.targetBalance,
                result.tradingFeePPM,
                result.targetAmount
            );

            result.sourceAmount = tradeAmountAndFee.amount;

            // ensure that the user has provided enough tokens to make the trade
            if (result.sourceAmount > result.limit) {
                revert InsufficientSourceAmount();
            }
        }

        result.tradingFeeAmount = tradeAmountAndFee.tradingFeeAmount;

        // sync the trading and staked balance
        result.sourceBalance += result.sourceAmount;
        result.targetBalance -= result.targetAmount;

        if (result.isSourceBNT) {
            result.stakedBalance += result.tradingFeeAmount;
        }

        _processNetworkFee(result);
    }
```

Listing 3.2: PoolCollection::_processTrade()

Moreover, when the subroutine `_processNetworkFee()` is invoked, the `network fee` amount is properly saved in `result.tradingFeeAmount` (line 1426). However, the target token balance needs to further reduce by the `network fee`. Namely, we need to add the following statement `result.targetBalance`

`-= targetNetworkFeeAmount` within the `if`-branch (lines 1425-1430).

```
1415    function _processNetworkFee(TradeIntermediateResult memory result) private view {
1416        uint32 networkFeePPM = _networkSettings.networkFeePPM();
1417        if (networkFeePPM == 0) {
1418            return;
1419        }
1420
1421        // calculate the target network fee amount and update the trading fee amount
            accordingly
1422        uint256 targetNetworkFeeAmount = MathEx.mulDivF(result.tradingFeeAmount,
            networkFeePPM, PPM_RESOLUTION);
1423        result.tradingFeeAmount -= targetNetworkFeeAmount;
1424
1425        if (!result.isSourceBNT) {
1426            result.networkFeeAmount = targetNetworkFeeAmount;
1427
1428            return;
1429        }
1430
1431        // trade the network fee (taken from the base token) to BNT
1432        result.networkFeeAmount = _tradeAmountAndFeeBySourceAmount(
1433            result.targetBalance,
1434            result.sourceBalance,
1435            0,
1436            targetNetworkFeeAmount
1437        ).amount;
1438
1439        // since we have received the network fee in base tokens and have traded them
            for BNT (so that the network fee
1440        // is always kept in BNT), we'd need to adapt the trading liquidity and the
            staked balance accordingly
1441        result.targetBalance += targetNetworkFeeAmount;
1442        result.sourceBalance -= result.networkFeeAmount;
1443        result.stakedBalance -= targetNetworkFeeAmount;
1444    }
```

Listing 3.3: PoolCollection::_processNetworkFee()

In the same vein, this issue is also present during the execution path where `isSourceBNT` is `true`. In particular, the current implementation unfortunately deducts the `network fee` twice from the target token balance.

Recommendation Revise the above swap logic to properly take into account the `network fee` in the target token balance.

Status This issue has been fixed in this commit: `e5b3a80`.

3.3 Improved Logic in Cancelling Pending Withdrawals

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PendingWithdrawals`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

For the same reason, it is always a best practice if we can avoid unnecessary storage reads. While reviewing the current pool token withdrawal logic, we notice the current implementation may be improved. In particular, we show below the related `_cancelWithdrawal()` function. This function emits a `WithdrawalCancelled` event that contains the reserve token `request.poolToken.reserveToken()` (line 266). It comes to our attention that instead of making the call to retrieve the reserve token, the reserve token information is readily available at `request.reserveToken` without the cost of making the extra inter-contract call and an extra storage read.

```
358    function _cancelWithdrawal(WithdrawalRequest memory request, uint256 id) private {
359        // remove the withdrawal request and its id from the storage
360        _removeWithdrawalRequest(request.provider, id);
361
362        // transfer the locked pool tokens back to the provider
363        request.poolToken.safeTransfer(request.provider, request.poolTokenAmount);
364
365        emit WithdrawalCancelled({
366            pool: request.poolToken.reserveToken(),
367            provider: request.provider,
368            requestId: id,
369            poolTokenAmount: request.poolTokenAmount,
370            reserveTokenAmount: request.reserveTokenAmount,
371            timeElapsed: _time() - request.createdAt
372        });
```

```
    }
```

Listing 3.4: PendingWithdrawals::_cancelWithdrawal()

Recommendation Improve the above `_cancelWithdrawal()` function to avoid unnecessary storage reads.

Status This issue has been fixed in this commit: `162c7c3`.

3.4 Proper TotalLiquidityUpdated Event Generation

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BNTPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `BNTPool` contract as an example. This contract is designed to manage the current pool liquidity. While examining the events that reflect the `liquidity` changes, we notice the emitted information in `TotalLiquidityUpdated` can be improved. Specifically, the emitted event includes the current pool token's total supply `poolTokenSupply`. However, the current implementation uses the `poolTokenSupply` variable to keep the old total supply before the `liquidity` change! To correct, we need to reflect the change in `poolTokenSupply`! Note the same issue is also applicable to another routine `renounceFunding()`.

```
    function requestFunding(
        bytes32 contextId,
        Token pool,
        uint256 bntAmount
    ) external onlyRoleMember(ROLE_FUNDING_MANAGER) poolWhitelisted(pool)
        greaterThanZero(bntAmount) {
        uint256 currentFunding = _currentPoolFunding[pool];
        uint256 fundingLimit = _networkSettings.poolFundingLimit(pool);
        uint256 newFunding = currentFunding + bntAmount;

        // verify that the new funding amount doesn't exceed the limit
```

```
        if (newFunding > fundingLimit) {
            revert FundingLimitExceeded();
        }

        // calculate the pool token amount to mint
        uint256 currentStakedBalance = _stakedBalance;
        uint256 poolTokenAmount;
        uint256 poolTokenTotalSupply = _poolToken.totalSupply();
        if (poolTokenTotalSupply == 0) {
            // if this is the initial liquidity provision - use a one-to-one pool token to BNT rate
            if (currentStakedBalance > 0) {
                revert InvalidStakedBalance();
            }

            poolTokenAmount = bntAmount;
        } else {
            poolTokenAmount = _underlyingToPoolToken(bntAmount, poolTokenTotalSupply,
                currentStakedBalance);
        }

        // update the staked balance
        uint256 newStakedBalance = currentStakedBalance + bntAmount;
        _stakedBalance = newStakedBalance;

        // update the current funding amount
        _currentPoolFunding[pool] = newFunding;

        // mint pool tokens to the protocol
        _poolToken.mint(address(this), poolTokenAmount);

        // mint BNT to the vault
        _bntGovernance.mint(address(_masterVault), bntAmount);

        emit FundingRequested({
            contextId: contextId,
            pool: pool,
            bntAmount: bntAmount,
            poolTokenAmount: poolTokenAmount
        });

        emit TotalLiquidityUpdated({
            contextId: contextId,
            poolTokenSupply: poolTokenTotalSupply,
            stakedBalance: newStakedBalance,
            actualBalance: _bnt.balanceOf(address(_masterVault))
        });
    }
```

Listing 3.5: `BNTPool::requestFunding()`

Recommendation Properly emit the `TotalLiquidityUpdated` event when the liquidity is updated.

Status This issue has been fixed in this commit: f3c4cc2.

3.5 Improved Initialization Logic in AutoCompoundingStakingRewards

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AutoCompoundingStakingRewards
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The Bancor V3 contract allows for lazy contract initialization, i.e., the initialization does not need to be performed inside the constructor at deployment. This feature is enabled by introducing the initializer() and onlyInitializing() modifiers. The initializer() protects an initializer function from being invoked twice, and the onlyInitializing() modifier protects an initialization function so that it can only be invoked by functions with the initializer() modifier, directly or indirectly. While examining the usage of these two modifiers, we notice the existence of abuse of the initializer() modifier, which needs to be corrected.

To elaborate, we show below the code snippet of the AutoCompoundingStakingRewards::initialize() routine. As the name indicates, it is an initialization function for the AutoCompoundingStakingRewards contract. This initialize() function is protected by the initializer() and it further invokes the subcalls to __AutoCompoundingStakingRewards_init(), etc. (line 135). It comes to our attention that the initializer() is also applied to __ReentrancyGuard_init()/__Upgradeable_init() (line 144 and line 145). As a result, the initialization will fail at the validation (line 53) in the Initializable::initializer(), because the _initializing/_initialized have both been set to true by the initializer() of AutoCompoundingStakingRewards::initialize(). To correct, we suggest to protect the subcalls with the onlyInitializing() modifier, as recommended by Openzeppelin: #3006.

```
134     function initialize() external initializer {
135         __AutoCompoundingStakingRewards_init();
136     }
137
138     // solhint-disable func-name-mixedcase
139
140     /**
141      * @dev initializes the contract and its parents
142      */
143     function __AutoCompoundingStakingRewards_init() internal initializer {
144         __ReentrancyGuard_init();
145         __Upgradeable_init();
```

```
        __AutoCompoundingStakingRewards_init_unchained();
    }

    /**
     * @dev performs contract-specific initialization
     */
    function __AutoCompoundingStakingRewards_init_unchained() internal initializer {}
```

Listing 3.6: `AutoCompoundingStakingRewards::initialize()`

```
modifier initializer() {
    // If the contract is initializing we ignore whether _initialized is set in order to
         support multiple
    // inheritance patterns, but we only do this in the context of a constructor,
        because in other contexts the
    // contract may have been reentered.
    require(_initializing ? _isConstructor() : !_initialized, "Initializable: contract
        is already initialized");

    bool isTopLevelCall = !_initializing;
    if (isTopLevelCall) {
        _initializing = true;
        _initialized = true;
    }

    _;

    if (isTopLevelCall) {
        _initializing = false;
    }
}
```

Listing 3.7: `Initializable::initializer()`

Recommendation Enforce the initialization-related subcalls with the `onlyInitializing` modifier.

Status This issue has been fixed in this commit: `75eac2b`.

3.6 Proper Removal Of _programByPool In terminateProgram()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: AutoCompoundingStakingRewards
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Bancor V3 protocol is architecturally designed to incentivize users. By design, the contract `AutoCompoundingStakingRewards` allows an entity i.e., `admin`, to dynamically add a rewarding program that basically distributes rewards for suggested pool token(s). Specifically, there is a routine `createProgram()` that is defined to add and apply the new program. Also, there is another `terminateProgram()` counterpart to terminate the program. While reviewing the program addition and termination logic, we notice the current implementation can be improved.

To elaborate, we show below the full implementation of the `terminateProgram()` routine. Note the a created program not only has the associated `_programs[pool]` details, but also the `_programByPool[pool]` mapping with the pool token. When a program is terminated, the current implementation only cleans up the associated `_programs[pool]` details, while leaving the associated `_programByPool[pool]` mapping intact.

```
    function terminateProgram(Token pool) external onlyAdmin {
        ProgramData memory p = _programs[pool];

        if (!_doesProgramExist(p)) {
            revert ProgramDoesNotExist();
        }

        delete _programs[pool];

        emit ProgramTerminated({ pool: pool, endTime: p.endTime, remainingRewards: p.
            remainingRewards });
    }
```

Listing 3.8: `AutoCompoundingStakingRewards::terminateProgram()`

Recommendation When a program is terminated, there is also a need to properly remove the associated `_programByPool[pool]` mapping.

Status This issue has been fixed in this commit: `d4d7c55`.

3.7 Proper Enforcement of depositingEnabled in PoolCollection

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PoolCollection`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Bancor V3` protocol is no exception. Specifically, if we examine the `PoolCollection` contract, it has defined a number of protocol-wide risk parameters, such as `tradingEnabled` and `depositingEnabled`. In the following, we show the corresponding routines that allow for their changes.

These protocol parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. However, our analysis shows that the `depositingEnabled` parameter is defined, but not enforced. Note this parameter defines whether a pool accepts the user deposits and there is a need to validate this configuration before a user deposit can be accepted.

```
    function enableDepositing(Token pool, bool status) external onlyOwner {
        Pool storage data = _poolStorage(pool);

        if (data.depositingEnabled == status) {
            return;
        }

        data.depositingEnabled = status;

        emit DepositingEnabled({ pool: pool, newStatus: status });
    }

    function setDepositLimit(Token pool, uint256 newDepositLimit) external onlyOwner {
        Pool storage data = _poolStorage(pool);

        uint256 prevDepositLimit = data.depositLimit;
        if (prevDepositLimit == newDepositLimit) {
            return;
        }

        data.depositLimit = newDepositLimit;

        emit DepositLimitUpdated({ pool: pool, prevDepositLimit: prevDepositLimit,
            newDepositLimit: newDepositLimit });
    }
```

Listing 3.9: `PoolCollection::enableDepositing()/setDepositLimit()`

Recommendation Validate and enforce any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status This issue has been fixed in this commit: `df19209`.

3.8 Proper Fee Collection in BancorNetwork::_tradeBNT()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BancorNetwork`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `Bancor V3` protocol is in essence a `DEX` that has the built-in support of swapping one token to another. And the `BancorNetwork` contract is the entry point for trading users. While reviewing the current logic of trade fee accounting, we notice the current implementation may reduce the trade fee attribution to the liquidity providers.

To elaborate, we show below the full implementation of the `_tradeBNT()` routine. This is a core function that is designed to perform a single hop between `BNT` and a base token trade by providing either the source or the target amount. However, if we pay attention to the intermediate call to `_bntPool.onFeesCollected()`, it attempts to notify the `BNT` pool on collected fees when the target token is `BNT`. It comes to our attention the collected fees are calculated as `tradeAmountsAndFee.tradingFeeAmount - tradeAmountsAndFee.networkFeeAmount` (line 1218), which should be `tradeAmountsAndFee.tradingFeeAmount`! The reason is that the `tradeAmountsAndFee.tradingFeeAmount` is already reduced with the `network fee`, i.e., `tradeAmountsAndFee.networkFeeAmount`. The current implementation may collect less trade fee, which leads to smaller return for liquidity providers.

```
1178     function _tradeBNT(
1179         bytes32 contextId,
1180         Token pool,
1181         bool isSourceBNT,
1182         TradeParams memory params,
1183         address trader
1184     ) private returns (TradeAmountAndNetworkFee memory) {
1185         TradeTokens memory tokens = isSourceBNT
1186             ? TradeTokens({ sourceToken: Token(address(_bnt)), targetToken: pool })
1187             : TradeTokens({ sourceToken: pool, targetToken: Token(address(_bnt)) });
1188 
1189         TradeAmountAndFee memory tradeAmountsAndFee = params.bySourceAmount
1190             ? _poolCollection(pool).tradeBySourceAmount(
1191                 contextId,
```

```
                tokens.sourceToken,
                tokens.targetToken,
                params.amount,
                params.limit
            )
            : _poolCollection(pool).tradeByTargetAmount(
                contextId,
                tokens.sourceToken,
                tokens.targetToken,
                params.amount,
                params.limit
            );

        // if the target token is BNT, notify the BNT pool on collected fees
        if (!isSourceBNT) {
            _bntPool.onFeesCollected(
                pool,
                tradeAmountsAndFee.tradingFeeAmount - tradeAmountsAndFee.
                    networkFeeAmount,
                true
            );
        }

        TradeAmounts memory tradeAmounts = params.bySourceAmount
            ? TradeAmounts({ sourceAmount: params.amount, targetAmount:
                tradeAmountsAndFee.amount })
            : TradeAmounts({ sourceAmount: tradeAmountsAndFee.amount, targetAmount:
                params.amount });

        emit TokensTraded({
            contextId: contextId,
            pool: pool,
            sourceToken: tokens.sourceToken,
            targetToken: tokens.targetToken,
            sourceAmount: tradeAmounts.sourceAmount,
            targetAmount: tradeAmounts.targetAmount,
            bntAmount: isSourceBNT ? tradeAmounts.sourceAmount : tradeAmounts.
                targetAmount,
            targetFeeAmount: tradeAmountsAndFee.tradingFeeAmount,
            bntFeeAmount: isSourceBNT ? tradeAmountsAndFee.networkFeeAmount :
                tradeAmountsAndFee.tradingFeeAmount,
            trader: trader
        });

        return
            TradeAmountAndNetworkFee({
                amount: tradeAmountsAndFee.amount,
                networkFeeAmount: tradeAmountsAndFee.networkFeeAmount
            });
    }
```

Listing 3.10: BancorNetwork::_tradeBNT()

Recommendation Revise the above logic to properly account for the trade fee collection in the BNT pool.

Status This issue has been fixed in this commit: 83c07a8.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the Bancor V3 protocol, there is a special admin account (with the ROLE_ADMIN). This admin account plays a critical role in governing and regulating the protocol-wide operations (e.g., assign other roles, configure various settings, and create incentive programs). It also has the privilege to control or govern the flow of assets within the protocol contracts. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
     function addTokenToWhitelist(Token token) external onlyAdmin {
        _addTokenToWhitelist(token);
    }

    function removeTokenFromWhitelist(Token token) external onlyAdmin {
        if (!_protectedTokenWhitelist.remove(address(token))) {
            revert DoesNotExist();
        }

        emit TokenRemovedFromWhitelist({ token: token });
    }

    function setFundingLimit(Token pool, uint256 amount) external onlyAdmin {
        _setFundingLimit(pool, amount);
    }

    function setMinLiquidityForTrading(uint256 amount) external onlyAdmin {
        uint256 prevMinLiquidityForTrading = _minLiquidityForTrading;
        if (_minLiquidityForTrading == amount) {
            return;
        }

        _minLiquidityForTrading = amount;

        emit MinLiquidityForTradingUpdated({ prevLiquidity: prevMinLiquidityForTrading,
            newLiquidity: amount });
```

```
    }
```

Listing 3.11: Example Privileged Operations in `NetworkSettings`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies that the admin key will be mitigated with a multisig account managed by trusted entities.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Bancor V3` protocol, which is a fully on-chain liquidity protocol and further introduces a new kind of composable single-sided pool token that only rises in relation to the staked asset. It also revises its tokenomics to enable a more cost-efficient system for `IL` protection and create greater deflationary pressure on `BNT`. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.