SMART CONTRACT AUDIT REPORT

for

BANCOR

Prepared By: Shuxiao Wang

PeckShield

February 27, 2021

Document Properties

Client	Bancor
Title	Smart Contract Audit Report
Target	StakingRewards
Version	1.0
Author	Xuxian Jiang
Auditors	Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 27, 2021	Xuxian Jiang	Final Release
1.0-rc	February 26, 2021	Xuxian Jiang	Release Candidate
0.2	February 21, 2021	Xuxian Jiang	Additional Findings #1
0.1	February 19, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
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's StakingRewards support, 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. BancorV2 further advances the DEX frontline by effectively mitigating the risk of impermanent loss for both stable and volatile tokens, providing liquidity with 100% exposure to a single reserve token, and offering a more efficient bonding curve that reduces slippage. This audit covers the StakingRewards functionality that allows for earning yields for staking users and incentivizing the community to further broaden user adoption. The goal is to create enough incentive to draw more liquidity into Bancor pools and attract more conversions in the market.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of StakingRewards

Item	Description
Issuer	Bancor
Website	http://bancor.network/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 27, 2021

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

- https://github.com/bancorprotocol/staking-rewards.git (b259ae3)

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

- https://github.com/bancorprotocol/staking-rewards.git (0e2ad6c)

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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.

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) [6], 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 1.3: 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

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 the staking support in bZx. 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	0	
Medium	2	■ ■
Low	1	■
Informational	2	■ ■
Total	5	

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 2 medium-severity vulnerabilities, and 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Lack of Overflow Validation in addPoolProgram()	Numeric Errors	Fixed
PVE-002	Informatinoal	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-003	Informatinoal	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-004	Low	Improved Logic in claimPendingRewards() And updateRewards()	Business Logic	Resolved
PVE-005	Medium	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed

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 Lack of Overflow Validation in addPoolProgram()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: StakingRewardsStore
- Category: Numeric Errors [5]
- CWE subcategory: CWE-190 [2]

Description

The StakingRewards protocol is architecturally designed to incentivize users. By design, the contract StakingRewardsStore allows an entity i.e., manager, to dynamically add a rewarding program that basically distributes rewards for suggested pool token(s). Specifically, there is a routine addPoolProgram() that is defined to add and apply the new program.

To elaborate, we show below the full implementation of the addPoolProgram() routine. This is a protected function that can only be invoked by the configured manager to specify the intended rewardRate as rewards within the incentivizing program.

```
    /**
     * @dev adds a program
     *
     * @param poolToken the pool token representing the rewards pool
     * @param reserveTokens the reserve tokens representing the liquidity in the pool
     * @param rewardShares reserve reward shares
     * @param endTime the ending time of the program
     * @param rewardRate the program's rewards rate per-second
     */
    function addPoolProgram(
        IDSToken poolToken,
        IERC20Token[2] calldata reserveTokens,
        uint32[2] calldata rewardShares,
        uint256 endTime,
        uint256 rewardRate
    ) external override onlyManager validAddress(address(poolToken)) {
```

```
        uint256 currentTime = time();

        addPoolProgram(poolToken, reserveTokens, rewardShares, currentTime, endTime, rewardRate);

        emit PoolProgramAdded(poolToken, currentTime, endTime, rewardRate);
    }
```

Listing 3.1: StakingRewardsStore::addPoolProgram()

However, a further analysis of the logic shows another related routine `verifyFullReward()`, which is responsible for calculating the maximum possible reward rate for each staked token and it is always invoked up-front before using the latest reward rate to ensure it is properly verified.

```
1136     function verifyFullReward(
1137         uint256 fullReward,
1138         IERC20Token reserveToken,
1139         PoolRewards memory poolRewardsData,
1140         PoolProgram memory program
1141     ) private pure {
1142         uint256 maxClaimableReward =
1143             (
1144                 program
1145                     .rewardRate
1146                     .mul(program.endTime.sub(program.startTime))
1147                     .mul(rewardShare(reserveToken, program))
1148                     .mul(MAX_MULTIPLIER)
1149                     .div(PPM_RESOLUTION)
1150                     .div(PPM_RESOLUTION)
1151             )
1152                 .sub(poolRewardsData.totalClaimedRewards);
1153
1154         // make sure that we aren't exceeding the full reward rate for any reason.
1155         require(fullReward <= maxClaimableReward, "ERR_REWARD_RATE_TOO_HIGH");
1156     }
```

Listing 3.2: StakingRewards:: verifyFullReward ()

A potential issue may surface if an oversized reward parameter `rewardRate` of the new program is applied. In particular, with the multiplication of four `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines $1144 - 1150$), especially when the `rewardRate` is largely controlled by an external entity, i.e., `manager`. An overflowed computation may revert ongoing transactions and potentially disable the entire protocol! Fortunately, the authentication check on the caller of `addPoolProgram()` greatly alleviates such concern. Currently, only the `manager` address is able to call.

Recommendation Apply necessary measures to mitigate the potential overflow risk in the incentivizer mechanism.

Status This issue has been fixed in this commit: `fa108f8`.

3.2 Improved Validation Of Function Arguments

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: StakingRewardsStore
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

The Bancor's StakingRewards implementation follows a widely-used pattern by isolating the state in a dedicated contract, i.e., StakingRewardsStore. With the state contract, it provides well-defined APIs to allow for safe and reliable state retrieval and update. In the following, we examine a specific setter routine setPoolsRewardData() that updates pool rewards data for multiple pools.

```
    /**
     * @dev seeds pool rewards data for multiple pools
     *
     * @param poolTokens pool tokens representing the rewards pool
     * @param reserveTokens reserve tokens representing the liquidity in the pool
     * @param lastUpdateTimes last update times (for both the network and reserve tokens)
     * @param rewardsPerToken reward rates per-token (for both the network and reserve tokens)
     * @param totalClaimedRewards total claimed rewards up until now (for both the network and reserve tokens)
     */
    function setPoolsRewardData(
        IDSToken[] calldata poolTokens,
        IERC20Token[] calldata reserveTokens,
        uint256[] calldata lastUpdateTimes,
        uint256[] calldata rewardsPerToken,
        uint256[] calldata totalClaimedRewards
    ) external onlySeeder {
        uint256 length = poolTokens.length;
        require(
            length == reserveTokens.length && length == lastUpdateTimes.length && length == rewardsPerToken.length,
            "ERR_INVALID_LENGTH"
        );

        for (uint256 i = 0; i < length; ++i) {
            IDSToken poolToken = poolTokens[i];
            _validAddress(address(poolToken));

            IERC20Token reserveToken = reserveTokens[i];
            _validAddress(address(reserveToken));

            PoolRewards storage data = _poolRewards[poolToken][reserveToken];
```

```
445             data.lastUpdateTime = lastUpdateTimes[i];
446             data.rewardPerToken = rewardsPerToken[i];
447             data.totalClaimedRewards = totalClaimedRewards[i];
448         }
449     }
```

Listing 3.3: StakingRewardsStore::setPoolsRewardData()

This routine essentially iterates the given `poolTokens` and updates related rewards for the participating pools (lines 445 – 447), including states such as `lastUpdateTime`, `rewardPerToken`, and `totalClaimedRewards`. Within the routine, it properly validates the given arguments in ensuring the given arrays have the same length. It turns out that the validation misses the length check on the last argument, i.e., `totalClaimedRewards`.

Recommendation Add the length check on all given arguments of `setPoolsRewardData`. An example revision is shown below:

```
        require(
            length == reserveTokens.length && length == lastUpdateTimes.length && length
                == rewardsPerToken.length && length == totalClaimedRewards.length,
            "ERR_INVALID_LENGTH"
        );
```

Listing 3.4: StakingRewardsStore::setPoolsRewardData()

Status This issue has been fixed in this commit: `cd71fcc`.

3.3 Inconsistency Between Document and Implementation

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StakingRewards`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

A few example comments can be found in line 120 of `StakingRewards::onAddingLiquidity()`, line 685 of `StakingRewards::storeRewards()`, and line 767 of `StakingRewards::rewardPerToken()` of the same contract, Using the `onAddingLiquidity()` routine as an example, the preceding function summary indicates that this callback routine should be called after the liquidity is added. However, our analysis shows that it is called before the liquidity is added.

```
    /**
     * @dev liquidity provision notification callback. The callback should be called *
        after* the liquidity is added in
     * the LP contract.
     *
     * @param provider the owner of the liquidity
     * @param poolAnchor the pool token representing the rewards pool
     * @param reserveToken the reserve token of the added liquidity
     */
    function onAddingLiquidity(
        address provider,
        IConverterAnchor poolAnchor,
        IERC20Token reserveToken,
        uint256, /* poolAmount */
        uint256 /* reserveAmount */
    ) external override onlyPublisher validExternalAddress(provider) {
        if (!_store.isReserveParticipating(IDSToken(address(poolAnchor)), reserveToken))
            {
            return;
        }

        updateRewards(provider, IDSToken(address(poolAnchor)), reserveToken,
            liquidityProtectionStats());
    }
```

Listing 3.5: StakingRewards::onAddingLiquidity()

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status This issue has been fixed in following commits: `c64a30a`, and `00f7ced`.

3.4 Improved Logic in claimPendingRewards() And updateRewards()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakingRewards`
- Category: Business Logic [4]
- CWE subcategory: N/A

Description

As mentioned in Section 3.1, StakingRewards supports different programs of incentives. And these different programs are facilitated with a number of helper routines. In the following, we examine two specific helper routines, i.e. `claimPendingRewards()` and `updateRewards()`. The first routine is used to claim pending rewards for a participating user (or provider) and the second routine is used to update internal states of current active programs.

To elaborate, we show below the implementation of a specific `claimPendingRewards()` routine. As mentioned earlier, this specific routine is used to claim specific provider's pending rewards for a specific list of participating pools. This routine further calls another internal helper that handles the claim for a specific participating pool. With the dynamic nature of accruing rewards, it is the likely that the given `maxAmount` may not be precise, which can easily revert the operation of `maxAmount = maxAmount.sub(poolReward)` (line 368). It happens when the remaining `maxAmount` is less than the current `poolReward` being examined.

```
342     /**
343      * @dev claims specific provider's pending rewards for a specific list of
            participating pools
344      *
345      * @param provider the owner of the liquidity
346      * @param poolTokens the list of participating pools to query
347      * @param maxAmount an optional bound on the rewards to claim (when partial claiming
             is required)
348      * @param lpStats liquidity protection statistics store
349      * @param resetStakingTime true to reset the effective staking time, false to keep
            it as is
350      *
351      * @return all pending rewards
352      */
353     function claimPendingRewards(
354         address provider,
355         IDSToken[] memory poolTokens,
356         uint256 maxAmount,
357         ILiquidityProtectionStats lpStats,
358         bool resetStakingTime
359     ) private returns (uint256) {
360         uint256 reward = 0;

362         uint256 length = poolTokens.length;
363         for (uint256 i = 0; i < length && maxAmount > 0; ++i) {
364             uint256 poolReward = claimPendingRewards(provider, poolTokens[i], maxAmount,
                     lpStats, resetStakingTime);
365             reward = reward.add(poolReward);

367             if (maxAmount != MAX_UINT256) {
368                 maxAmount = maxAmount.sub(poolReward);
369             }
370         }

372         return reward;
373     }
```

Listing 3.6: StakingRewards::claimPendingRewards()

For the next helper routine `updateRewards()`, we notice an optimization that can make early exist when the pool is no longer participating (line 907).

```
893     /**
894      * @dev updates pool and provider rewards. this function is called during every
           liquidity changes
895      *
896      * @param provider the owner of the liquidity
897      * @param poolToken the pool token representing the rewards pool
898      * @param reserveToken the reserve token representing the liquidity in the pool
899      * @param lpStats liquidity protection statistics store
900      */
901     function updateRewards(
902         address provider,
903         IDSToken poolToken,
904         IERC20Token reserveToken,
905         ILiquidityProtectionStats lpStats
906     ) private returns (PoolRewards memory, ProviderRewards memory) {
907         PoolProgram memory program = poolProgram(poolToken);

909         // calculate the new reward rate per-token and update it in the store.
910         PoolRewards memory poolRewardsData = poolRewards(poolToken, reserveToken);

912         // rewardPerToken must be calculated with the previous value of lastUpdateTime.
913         poolRewardsData.rewardPerToken = rewardPerToken(poolToken, reserveToken,
                poolRewardsData, program, lpStats);
914         poolRewardsData.lastUpdateTime = Math.min(time(), program.endTime);

916         ...
917     }
```

Listing 3.7: StakingRewards::updateRewards()

Recommendation Revise current execution logic of `stake()` to defensively calculate the share amount when the pool is being initialized.

Status This issue has been confirmed. However, the issue will not happen in this specific implementation, since in the internal `claimPendingRewards()` helper function, if the reward is higher than `maxAmount` - it'll be set to it. So in the worst case `maxAmount` should equal to the value of the `poolReward` variable.

3.5 Accommodation of approve() Idiosyncrasies

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `StakingRewards`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.8: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. An example is shown below. It is in the `stakeRewards()` routine that is designed to stake the rewards back into the pool. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function stakeRewards(
        address provider,
        IDSToken[] memory poolTokens,
        uint256 maxAmount,
        IDSToken newPoolToken,
        ILiquidityProtectionStats lpStats
    ) private returns (uint256, uint256) {
        uint256 amount = claimPendingRewards(provider, poolTokens, maxAmount, lpStats,
            false);
        if (amount == 0) {
            return (amount, 0);
        }

        // approve the LiquidityProtection contract to pull the rewards.
        ILiquidityProtection liquidityProtection = liquidityProtection();
        _networkToken.safeApprove(address(liquidityProtection), amount);

        // mint the reward tokens directly to the staking contract, so that the
            LiquidityProtection could pull the
        // rewards and attribute them to the provider.
        _networkTokenGovernance.mint(address(this), amount);

        uint256 newId =
            liquidityProtection.addLiquidityFor(provider, newPoolToken, IERC20Token(
                address(_networkToken)), amount);

        // please note, that in order to incentivize staking, we won't be updating the
            time of the last claim, thus
        // preserving the rewards bonus multiplier.

        emit RewardsStaked(provider, newPoolToken, amount, newId);

        return (amount, newId);
    }
```

Listing 3.9: StakingRewards::stakeRewards()

Note that the accommodation of the `approve()` idiosyncrasy is necessary to ensure a smooth stake operation. Otherwise, the stake attempt with inconsistent token contracts may always be reverted.

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status This issue has been fixed in this commit: `d5d372c`.

4 Conclusion

In this audit, we have analyzed the `Bancor`'s `StakingRewards` design and implementation. The system presents a unique enhancement to current `Bancor` protocol. 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

[5] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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