SMART CONTRACT AUDIT REPORT

for

BANCOR

Prepared By: Shuxiao Wang

Hangzhou, China
October 11, 2020

Document Properties

Client	Bancor
Title	Smart Contract Audit Report
Target	Governance and Liquidity Protection
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 11, 2020	Xuxian Jiang	Final Release
0.3	October 7, 2020	Xuxian Jiang	Additional Findings #2
0.2	October 2, 2020	Xuxian Jiang	Additional Findings #1
0.1	September 25, 2020	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 Bancor's **Governance and Liquidity Protection** design document and related smart contract source code, we in the report outline 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 Bancor's governance and liquidity protection 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 BancorV2 advances the DEX frontline in further 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 new BancorV2 modules that implement the features of its own governance and liquidity protection.

The basic information of Governance and Liquidity Protection is as follows:

Table 1.1: Basic Information of Governance and Liquidity Protection

Item	Description
Issuer	Bancor
Website	http://bancor.network/
Audit Modules	Governance and Liquidity Protection
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 11, 2020

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. For the `liquidity-protection` repository, it contains a number of sub-directories (e.g., `bancox`, `converter`, and `liquidity-protection`) and this audit covers only the `liquidity-protection` sub-directory.

- https://github.com/bancorprotocol/gov-contracts.git (2a20137)
- https://github.com/bancorprotocol/liquidity-protection.git (4ce6834)

1.2 About PeckShield

PeckShield Inc. [18] 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 [13]:

- 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 classified into four categories accordingly, 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) [12], 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.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 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.

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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the Bancor's governance subsystem and its new liquidity protection feature. 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	2
Informational	3
Total	8

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 that 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, 2 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Governance and Liquidity Protection Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Flashloan-Assisted Sandwich Attacks To Foil Proposals	Business Logics	Fixed
PVE-002	Informational	Incompatibility with Deflationary/Rebasing Tokens	Business Logics	Confirmed
PVE-003	Informational	Missed Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-004	Medium	Possible Front-Running To Block Proposal Execution	Time and State	Fixed
PVE-005	Low	Inconsistent Calculation on Quorum Satisfaction	Coding Practices	Fixed
PVE-006	Medium	Unintended Removal of Voters' Stakes in revokeVotes()	Business Logics	Fixed
PVE-007	Low	Improved Verification of Matching IDs in unprotectLiquidity()	Security Features	Fixed
PVE-008	Informational	Optimization in removeLiquidityReturn()	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Flashloan-Assisted Sandwich Attacks To Foil Proposals

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: BancorGovernance
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

In Bancor, the governance subsystem works by requiring voters to stake their assets (i.e., gBNT). These staked assets represent voting powers and will be locked for a predefined lock duration if used for voting. A proposal will not be considered pass if the number of voters does not meet the required quorum.

To elaborate, we show the code snippet of the voteFor() routine. As the name indicates, it is used to vote in favor of the target proposal. For every incoming vote, either For or Against, the current quorum is calculated in the calculateQuorumRatio() routine.

```
    /**
     * @notice votes for a proposal
     *
     * @param _id id of the proposal to vote for
     */
    function voteFor(uint256 _id) public onlyStaker proposalNotEnded(_id) {
        // mark sender as voter
        voters[msg.sender] = true;

        // get against votes for this sender
        uint256 votesAgainst = proposals[_id].votesAgainst[msg.sender];
        // do we have against votes for this sender?
        if (votesAgainst > 0) {
            // yes, remove the against votes first
            proposals[_id].totalVotesAgainst = proposals[_id].totalVotesAgainst.sub(
                votesAgainst);
```

```
477             proposals[_id].votesAgainst[msg.sender] = 0;
478         }

480         // calculate voting power in case voting for twice
481         uint256 vote = votesOf(msg.sender).sub(proposals[_id].votesFor[msg.sender]);

483         // increase total for votes of the proposal
484         proposals[_id].totalVotesFor = proposals[_id].totalVotesFor.add(vote);
485         // set for votes to the votes of the sender
486         proposals[_id].votesFor[msg.sender] = votesOf(msg.sender);
487         // update total votes available on the proposal
488         proposals[_id].totalVotesAvailable = totalVotes;
489         // recalculate quorum based on overall votes
490         proposals[_id].quorum = calculateQuorumRatio(_id);
491         // lock sender
492         voteLocks[msg.sender] = voteLock.add(block.number);

494         // emit vote event
495         emit Vote(_id, msg.sender, true, vote);
496     }
```

Listing 3.1: BancorGovernance.sol

Our analysis shows that the way `calculateQuorumRatio()` calculates the quorum is based on two numbers. The first number is in essence the current votes on the proposal, i.e., `totalProposalVotes`; and the second number is the total number of votes staked in the system, i.e., `totalVotes`. It is important to point out that every `stake()` will increase `totalVotes` while every `unstake()` will decrease `totalVotes`, no matter whether the votes are casted or not.

```
    function calculateQuorumRatio(uint256 _id) internal view returns (uint256) {
        // calculate overall votes
        uint256 totalProposalVotes = proposals[_id].totalVotesFor.add(
            proposals[_id].totalVotesAgainst
        );

        return totalProposalVotes.mul(10000).div(totalVotes);
    }
```

Listing 3.2: BancorGovernance.sol

Unfortunately, the total number of votes in the system are not subject to the predefined lockup period. As a result, a malicious attack can be possibly arranged by sandwiching a `voteFor()` transaction with a preceding one and a tailgating one. The preceding transaction can be a flashloan-assisted `stake()` to dramatically increase `totalVotes` and the tailgating one is the `unstake()` counterpart that basically returns back the flashloan. The purpose here is to only increase `totalVotes` for the sandwiched `voteFor()` such that the proposal being voted always has an extremely low quorum, i.e., `proposals[_id].quorum = calculateQuorumRatio(_id)` (line 490).

```
    function stake(uint256 _amount) public {
        require(_amount > 0, "ERR_STAKE_ZERO");
```

```
        // increase vote power
        votes[msg.sender] = votesOf(msg.sender).add(_amount);
        // increase total votes
        totalVotes = totalVotes.add(_amount);
        // transfer tokens to this contract
        govToken.safeTransferFrom(msg.sender, address(this), _amount);

        // emit staked event
        emit Staked(msg.sender, _amount);
    }
```

Listing 3.3: BancorGovernance.sol

Instead of sandwiching other's legitimate `voteFor()` transactions, the malicious actor can simply vote herself and sandwich her voting transaction in a similar way. By doing so, the malicious actor can foil any submitted proposal.

Recommendation Enforce the predefined lockup period for the staked assets to defeat possible flashloans.

Status The issue has been confirmed and accordingly fixed by enforcing the predefined lock period for certain portion of staked assets. The fixup chooses 10% of staked assets for the lockup and the commit can be found below: `fa4125483241a02c09dbb64fa78106ea3eacedf5`.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BancorGovernance`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

The `BancorGovernance` contract behaves as the main entry for interaction with voting users. In particular, one entry routine, i.e., `stake()`, accepts user stakes of supported assets (e.g., `gBNT`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `BancorGovernance` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    /**
     * @notice stakes vote tokens
     *
```

```
     * @param _amount amount of vote tokens to stake
     */
    function stake(uint256 _amount) public {
        require(_amount > 0, "ERR_STAKE_ZERO");

        // increase vote power
        votes[msg.sender] = votesOf(msg.sender).add(_amount);
        // increase total votes
        totalVotes = totalVotes.add(_amount);
        // transfer tokens to this contract
        govToken.safeTransferFrom(msg.sender, address(this), _amount);

        // emit staked event
        emit Staked(msg.sender, _amount);
    }
```

Listing 3.4: BancorGovernance.sol

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `stake()` and `unstake()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer()` or `transferFrom()` will always result in the full transfer, we need to ensure the increased or decreased amount in the `BancorGovernance` before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into the governance subsystem. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

We emphasize that the current deployment is safe since it only supports `gBNT` for stakes and `gBNT` is not deflationary or rebasing. However, the current code implementation is generic in supporting various tokens and there is a need to highlight the possible pitfall from the audit perspective.

Recommendation Since this deployment uses the `gBNT` as the staking asset, there is no need to address this issue. However, if current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is widely-adopted

USDT.

Status As mentioned above, with gBNT as the staking asset, there is no need to address this issue.

3.3 Missed Sanity Checks For System Parameters

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: BancorGovernance
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [4]

Description

The governance subsystem in Bancor has a few system-wide parameters that can be dynamically adjusted. For example, quorum specifies the quorum needed for proposals to pass; voteMinimum indicates the needed votes for a proposer to submit a proposal; voteDuration controls the default voting duration of a submitted proposal; and voteLock requires the post-vote lock duration for the staked assets. Naturally, these parameters have their corresponding update routines, i.e., setQuorum(), setVoteMinimum(), setVoteDuration(), and setVoteLock().

While reviewing these system parameters, our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks and emitting relevant events to notify off-chain analytics and reporting tools.

```
    /**
     * @notice updates the quorum needed for proposals to pass
     *
     * @param _quorum required quorum
     */
    function setQuorum(uint256 _quorum) public ownerOnly {
        quorum = _quorum;
    }

    /**
     * @notice updates the required votes needed to propose
     *
     * @param _voteMinimum required minimum votes
     */
    function setVoteMinimum(uint256 _voteMinimum) public ownerOnly {
        voteMinimum = _voteMinimum;
    }

    /**
     * @notice updates the proposals voting duration
```

```
     *
     * @param _voteDuration vote duration
     */
    function setVoteDuration(uint256 _voteDuration) public ownerOnly {
        voteDuration = _voteDuration;
    }

    /**
     * @notice updates the post vote lock duration
     *
     * @param _voteLock vote lock
     */
    function setVoteLock(uint256 _voteLock) public ownerOnly {
        voteLock = _voteLock;
    }
```

Listing 3.5: BancorGovernance.sol

Recommendation Validate the given arguments before updating these system-wide parameters and emit relevant events to notify off-chain analytics tools.

Status The issue has been fixed by this commit: `c7b8ac53fb1dc7e7e122474df8a6956e5e871184`.

3.4 Possible Front-Running To Block Proposal Execution

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BancorGovernance`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [6]

Description

As mentioned in Section 3.1, the governance subsystem in Bancor specifies the entire life-cycle of a proposal. A proposal, if successfully passed, will lead to its activation in triggering the enclosed `executor`.

To elaborate, we show below the code snippet of the `execute()` routine that is responsible to trigger the proposal execution after necessary validation. However, we notice that an earlier call before invoking `executor` is made to `tallyVotes()`.

```
    function execute(uint256 _id) public proposalEnded(_id) {
        // get voting info of proposal
        (uint256 forRatio, uint256 againstRatio, uint256 quorumRatio) = proposalStats(
            _id);
        // check proposal state
        require(proposals[_id].quorumRequired < quorumRatio, "ERR_NO_QUORUM");
```

```

        // tally votes
        tallyVotes(_id);
        // do execution on the contract to be executed
        IExecutor(proposals[_id].executor).execute(_id, forRatio, againstRatio,
            quorumRatio);

        // emit proposal executed event
        emit ProposalExecuted(_id, proposals[_id].executor);
    }
```

Listing 3.6: BancorGovernance.sol

This `tallyVotes()` routine basically closes the proposal (line 417) and emits the `ProposalFinished` event. This execution logic seems sound and necessary.

```
401     /**
402      * @notice tallies votes of proposal with given id
403      *
404      * @param _id id of the proposal to tally votes for
405      */
406     function tallyVotes(uint256 _id) public proposalEnded(_id) {
407         // get voting info of proposal
408         (uint256 forRatio, uint256 againstRatio, ) = proposalStats(_id);
409         // assume we have no quorum
410         bool quorumReached = false;
411         // do we have a quorum?
412         if (proposals[_id].quorum >= proposals[_id].quorumRequired) {
413             quorumReached = true;
414         }
415
416         // close proposal
417         proposals[_id].open = false;
418
419         // emit proposal finished event
420         emit ProposalFinished(_id, forRatio, againstRatio, quorumReached);
421     }
```

Listing 3.7: BancorGovernance.sol

However, our further analysis shows that current execution logic suffers from a front-running attack. In particular, upon observing the `execute()` transaction, a front-runner can arrange another transaction to invoke `tallyVotes()` on the same proposal. By doing so, the front-runner can immediately close the proposal before `execute()` is invoked. Once the proposal is closed, the `execute()` transaction will simply be reverted because of the proposal status check in the `proposalEnded(_id)` modifier (line 219).

```
217     modifier proposalEnded(uint256 _id) {
218         require(proposals[_id].start > 0 && proposals[_id].start < block.number, "
            ERR_NO_PROPOSAL");
219         require(proposals[_id].open, "ERR_NOT_OPEN");
```

```
        require(proposals[_id].end < block.number, "ERR_NOT_ENDED");
        _;
    }
```

Listing 3.8: BancorGovernance.sol

Recommendation Develop an effective mitigation to the above front-running attack to ensure normal proposal execution.

Status The issue has been fixed by this commit: `c7b8ac53fb1dc7e7e122474df8a6956e5e871184`. The team has expanded the proposal status set by including an additional separate flag to indicate whether the proposal has been executed or not.

3.5 Inconsistent Calculation on Quorum Satisfaction

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BancorGovernance`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1099 [3]

Description

The quorum calculation is critical to determine whether a proposal is passed or not. However, for the same quorum calculation, we notice unnecessary discrepancy in determining the result of a proposal. Specifically, the discrepancy stems from two related functions, i.e., `execute()` and `tallyVotes()`.

```
    function execute(uint256 _id) public proposalEnded(_id) {
        // get voting info of proposal
        (uint256 forRatio, uint256 againstRatio, uint256 quorumRatio) = proposalStats(
            _id);
        // check proposal state
        require(proposals[_id].quorumRequired < quorumRatio, "ERR_NO_QUORUM");

        // tally votes
        tallyVotes(_id);
        // do execution on the contract to be executed
        IExecutor(proposals[_id].executor).execute(_id, forRatio, againstRatio,
            quorumRatio);

        // emit proposal executed event
        emit ProposalExecuted(_id, proposals[_id].executor);
    }
```

Listing 3.9: BancorGovernance.sol

For comparison, we show the `execute()` routine above and the `tallyVotes()` routine below. The above case considers the proposal quorum is reached if `proposals[_id].quorumRequired<quorumRatio` while the below case considers the result based on `proposals[_id].quorumRequired<=proposals[_id].quorum`. In other words, a discrepancy occurs when `proposals[_id].quorum == proposals[_id].quorumRequired`.

```
    function tallyVotes(uint256 _id) public proposalEnded(_id) {
        // get voting info of proposal
        (uint256 forRatio, uint256 againstRatio, ) = proposalStats(_id);
        // assume we have no quorum
        bool quorumReached = false;
        // do we have a quorum?
        if (proposals[_id].quorum >= proposals[_id].quorumRequired) {
            quorumReached = true;
        }

        // close proposal
        proposals[_id].open = false;

        // emit proposal finished event
        emit ProposalFinished(_id, forRatio, againstRatio, quorumReached);
    }
```

Listing 3.10: BancorGovernance.sol

Recommendation Be consistent in determining whether a proposal is passed by resolving the above discrepancy.

Status The issue has been fixed by this commit: `c7b8ac53fb1dc7e7e122474df8a6956e5e871184`.

3.6 Unintended Removal of Voters' Stakes in revokeVotes()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BancorGovernance`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

The `BancorGovernance` contract contains a function named `revokeVotes()` that allows a voter to revoke her votes. However, the `revokeVotes()` function not only revokes the status of being a voter, but clears the internal recorded amount of staked assets. This seems an unintended behavior as the votes being revoked should not mean the staked assets are also removed. In current implementation, the voters will not get those staked assets back and the removed assets will be locked forever in the contract. And there is no way to recover these locked assets.

```
    /**
     * @notice revokes votes
     */
    function revokeVotes() public onlyVoter {
        voters[msg.sender] = false;
        totalVotes = totalVotes.sub(votes[msg.sender]);

        // emit vote revocation event
        emit VotesRevoked(msg.sender, votesOf(msg.sender), totalVotes);
        votes[msg.sender] = 0;
    }
```

Listing 3.11: BancorGovernance.sol

Recommendation Revise the `revokeVotes()` logic by returning back the staked assets back to the voter or recovering the removed assets for a better use.

Status The issue has been fixed by this commit: `c7b8ac53fb1dc7e7e122474df8a6956e5e871184`. The team decides to remove this function.

3.7 Improved Verification of Matching IDs in unprotectLiquidity()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LiquidityProtection`
- Category: Security Features [8]
- CWE subcategory: CWE-284 [5]

Description

The BancorV2 protocol proposes an interesting feature, i.e., `liquidity protection`, which offers protection against so-called impermanent loss. The impermanent loss essentially reflects the difference between holding an asset versus providing liquidity (e.g., in a DEX) and is typically a temporary loss of funds from providing liquidity. This feature addresses a long-due issue to better protect the interests of liquidity providers and is considered essential for wide adoption.

The implementation of `liquidity protection` involves the support of a number of related routines, i.e., `protectLiquidity()`, `unprotectLiquidity()`, `addLiquidity()`, and `removeLiquidity()`. While reviewing these functions, we notice a potential issue in `unprotectLiquidity()` whose purpose is to cancel a pair of protections created with an earlier `protectLiquidity()`.

To elaborate, we show below the `unprotectLiquidity()` routine. Its execution logic is rather straightforward in firstly validating the provided pair of protection `IDs` and then removing them from the storage (via `removeProtectedLiquidity()`).

```
450     function unprotectLiquidity(uint256 _id1, uint256 _id2) external protected {
451         require(_id1 != _id2, "ERR_SAME_ID");
452 
453         ProtectedLiquidity memory liquidity1 = protectedLiquidity(_id1);
454         ProtectedLiquidity memory liquidity2 = protectedLiquidity(_id2);
455 
456         // verify input & permissions
457         require(liquidity1.owner == msg.sender && liquidity2.owner == msg.sender, "
                ERR_ACCESS_DENIED");
458 
459         // verify that the two protections were added together (using 'protect')
460         require(
461             liquidity1.poolToken == liquidity2.poolToken &&
462             liquidity1.reserveToken != liquidity2.reserveToken &&
463             liquidity1.timestamp == liquidity2.timestamp &&
464             liquidity1.poolAmount <= liquidity2.poolAmount.add(1) &&
465             liquidity2.poolAmount <= liquidity1.poolAmount.add(1),
466             "ERR_PROTECTIONS_MISMATCH");
467 
468         // burn the governance tokens from the caller
469         govToken.destroy(msg.sender, liquidity1.reserveToken == networkToken ?
                liquidity1.reserveAmount : liquidity2.reserveAmount);
470 
471         // remove the protected liquidities from the store
472         store.removeProtectedLiquidity(_id1);
473         store.removeProtectedLiquidity(_id2);
474 
475         // transfer the pool tokens back to the caller
476         store.withdrawTokens(liquidity1.poolToken, msg.sender, liquidity1.poolAmount.add
                (liquidity2.poolAmount));
477     }
```

Listing 3.12: LiquidityProtection.sol

However, the validation checks are not sufficient. Imagine a scenario when an user creates two pairs of protections: The first pair has two IDs: `ID1-1` and `ID1-2`; and the second pair has `ID2-1` and `ID2-2`. For simplicity, we assume each ID shares the same `poolAmount` and the protections of `ID1-2` and `ID2-2` use `BNT` as their `networkToken`. With that, when the user may invoke `unprotectLiquidity(ID1-1, ID2-1)`, though the given `ID1-1` and `ID2-1` are not part of the same pair, they still successfully pass the current validation checks (lines 457 – 466). This is certainly not unintended behavior.

Recommendation Apply additional sanity checks in ensuring one of the matching IDs is `networkToken`.

Status The issue has been fixed by this commit: `e2f7a2ed4a5c1e218e510f0a694e5a38f5751397`.

3.8 Optimization in removeLiquidityReturn()

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LiquidityProtection`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [2]

Description

Following the discussions in Section 3.7, we continue our analysis in the `liquidity protection` feature. When reviewing the internal `removeLiquidityReturn()` routine, we notice a redundant computation that can be optimized. To elaborate, we show below the code snippet of the `removeLiquidityReturn()` routine.

```
814     function removeLiquidityReturn(
815         IDSToken _poolToken,
816         IERC20Token _reserveToken,
817         uint256 _poolAmount,
818         uint256 _reserveAmount,
819         Fraction memory _addRate,
820         Fraction memory _removeRate,
821         uint256 _addTimestamp,
822         uint256 _removeTimestamp)
823         internal view returns (uint256)
824     {
825         // get the adjusted amount of pool tokens based on the exposure and rate changes
826         uint256 outputAmount = adjustedAmount(_poolToken, _reserveToken, _poolAmount,
            _addRate, _removeRate);
827
828         // calculate the protection level
829         Fraction memory level = protectionLevel(_addTimestamp, _removeTimestamp);
830
831         // no protection, return the amount as is
832         if (level.n == 0) {
833             return outputAmount;
834         }
835
836         // protection is in effect, calculate loss / compensation
837         Fraction memory loss = impLoss(_addRate, _removeRate);
838         (uint256 compN, uint256 compD) = Math.reducedRatio(loss.n.mul(level.n), loss.d.
            mul(level.d), MAX_UINT128);
839         return outputAmount.mul(compD).add(_reserveAmount.mul(compN)).div(compD);
840     }
```

Listing 3.13: LiquidityProtection.sol

If we examine the code at line 839, i.e., `outputAmount.mul(compD).add(_reserveAmount.mul(compN)).div(compD)`, the calculated amount can be simplified as `outputAmount.add(_reserveAmount.mul(compN)`

.div(compD)). The reason is that the first `mul(compD)` will be immediately canceled out by the following `div(compD)`.

Recommendation Optimize the `removeLiquidityReturn()` routine as follows.

```
    function removeLiquidityReturn(
        IDSToken _poolToken,
        IERC20Token _reserveToken,
        uint256 _poolAmount,
        uint256 _reserveAmount,
        Fraction memory _addRate,
        Fraction memory _removeRate,
        uint256 _addTimestamp,
        uint256 _removeTimestamp)
        internal view returns (uint256)
    {
        // get the adjusted amount of pool tokens based on the exposure and rate changes
        uint256 outputAmount = adjustedAmount(_poolToken, _reserveToken, _poolAmount,
            _addRate, _removeRate);

        // calculate the protection level
        Fraction memory level = protectionLevel(_addTimestamp, _removeTimestamp);

        // no protection, return the amount as is
        if (level.n == 0) {
            return outputAmount;
        }

        // protection is in effect, calculate loss / compensation
        Fraction memory loss = impLoss(_addRate, _removeRate);
        (uint256 compN, uint256 compD) = Math.reducedRatio(loss.n.mul(level.n), loss.d.
            mul(level.d), MAX_UINT128);
        return outputAmount.add(_reserveAmount.mul(compN).div(compD));
    }
```

Listing 3.14: LiquidityProtection.sol

Status The issue has been fixed by this commit: `184ffb8eba6e4066911bd7484070aa4195ada22c`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Bancor's governance subsystem and its new liquidity protection feature. The system presents a unique offering in current DEX ecosystem with the support of its own governance and liquidity protection. We are impressed by the design and implementation, especially the underlying thinkings and efforts in reducing slippage and ensuring liquidity protection. The current code base is well organized and those identified issues are promptly confirmed and fixed.

As a final precaution, 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [14, 15, 16, 17, 19].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [20] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

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

[3] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

[5] MITRE. CWE-284: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html.

[6] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[11] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

[14] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[15] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[16] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[17] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[19] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[20] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.