Security Audit Report for BaconCoin Contracts

Date: May 8, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.05.08 16:40:17 +08'00'

Contents

1 Introduction **1**
1.1 About Target Contracts . . . 1
1.2 Disclaimer . . . 1
1.3 Procedure of Auditing . . . 1
1.3.1 Software Security . . . 2
1.3.2 DeFi Security . . . 2
1.3.3 NFT Security . . . 2
1.3.4 Additional Recommendation . . . 2
1.4 Security Model . . . 3

2 Findings **4**
2.1 Software Security . . . 5
2.1.1 Reentrancy in `PoolStaking.unstake` function . . . 5
2.1.2 Lack of access control in airdrop contract . . . 6
2.1.3 Potential loss in sending ERC-777 tokens . . . 6
2.1.4 Potential DoS caused by insufficient gas . . . 7
2.1.5 Inconsistent decimal usage in `PoolStaking.distribute` . . . 8
2.1.6 No `delegate` function implemented in BaconCoin contract . . . 10
2.1.7 Unnecessary usage of ERC-777 standard . . . 10
2.2 DeFi Security . . . 10
2.2.1 Risk-free interest rate reduction by reentrancy . . . 10
2.2.2 Incorrect interest rate divisor in `PoolCore.getLoanAccruedInterest` . . . 11
2.2.3 No liquidation logic provided . . . 12
2.2.4 Potential centrality problem . . . 12
2.2.5 Potential logic problem in `PoolStaking.withdraw` . . . 13
2.2.6 Potential incorrect logic in `PoolStaking.distribute` . . . 14
2.3 Additional Recommendation . . . 15
2.3.1 Refactor loops to mappings for gas optimization . . . 15
2.3.2 Remove unused variables and functions . . . 15
2.3.3 Fix wrong variable used in `PoolStaking.distribute` . . . 15
2.3.4 Remove redundant logic in `PoolStaking.distribute` . . . 16

Report Manifest

Item	Description
Client	LoanSnap
Target	BaconCoin Contracts

Version History

Version	Date	Description
1.0	May 8, 2022	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values of the repo [1] during the audit are shown in the following. Note that, we did **NOT** audit all the modules in the repository. Only the newest version of the contracts are within audit scope.

Project		Commit SHA
BaconCoin	Version 1	1a3f2091642bdf5ea4a7ffab7d386f1efef22128
	Version 2	bc8d8be64f1c114b41ad286024888a80ae513aef

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report do not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.

[1] https://github.com/loansnap/HomeDAO

- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

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

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found `thirteen` potential issues and `four` recommendations in the smart contracts, as follows:

- High Risk: 2
- Medium Risk: 5
- Low Risk: 6
- Recommendations: 4

ID	Severity	Description	Category	Status
1	High	Reentrancy in `PoolStaking.unstake` function	Software security	Fixed
2	Medium	Lack of access control in airdrop contract	Software security	Fixed*
3	Medium	Potential loss in sending ERC-777 tokens	Software security	Fixed
4	High	Potential DoS caused by insufficient gas	Software security	Fixed
5	Medium	Inconsistent decimal usage in `PoolStaking.distribute`	Software security	Fixed
6	Low	No `delegate` function implemented in BaconCoin contract	Software security	Fixed
7	Low	Unnecessary usage of ERC-777 standard	Software security	Fixed
8	Low	Risk-free interest rate reduction by reentrancy	DeFi security	Fixed*
9	Low	Incorrect interest rate divisor in `PoolCore.getLoanAccruedInterest`	DeFi security	Fixed
10	Medium	No liquidation logic provided	DeFi security	Acknowledged
11	Medium	Potential centrality problem	DeFi security	Confirmed
12	Low	Potential logic problem in `PoolStaking.withdraw`	DeFi security	Acknowledged
13	Low	Potential incorrect logic in `PoolStaking.distribute`	DeFi security	Fixed
14	-	Refactor loops to mappings for gas optimization	Recommendation	Fixed
15	-	Remove unused variables and functions	Recommendation	Fixed
16	-	Fix wrong variable used in `PoolStaking.distribute`	Recommendation	Fixed
17	-	Remove redundant logic in `PoolStaking.distribute`	Recommendation	Fixed

* This issue is NOT fixed by modifying the smart contract(s) directly, please refer to the *Feedback from the Project* of the issue for detailed description.

The details are provided in the following sections.

2.1 Software Security

2.1.1 Reentrancy in `PoolStaking.unstake` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In BaconCoin, liquidity providers (LPs) are allowed to stake the LP tokens received from `PoolCore` into a `PoolStaking` contract and receive BaconCoin rewards. Specifically, LPs can call the `stake` and `lendAndStake` functions of `PoolCore` to stake and transfer LP tokens to the `PoolStaking` contract. LPs are then allowed to call `PoolStaking.unstake` to unstake the staked LP tokens. Then after a locking period, the LP tokens can be redeemed by calling the `PoolStaking.withdraw` function. BaconCoin tokens are distributed in each call to `stake`, `unstake` and `distribute`.

However, the `unstake` function is subject to reentrancy attacks, where the call to `distribute` can re-enter the `unstake` function because the BaconCoin is an ERC-777 token. It means that by utilizing reentrancy to the `unstake` function, the `userToUnstake` state variable is accumulated and a malicious actor can accumulate the amount he can unstake, and eventually drain all the LP tokens in the `PoolStaking` contract.

```
function unstake(uint256 amount) public returns (uint256) {
    require(userStaked[msg.sender] >= amount, "not enough staked");

    uint256 previousPending = userToUnstake[msg.sender].amount;
    userToUnstake[msg.sender] = UnstakeRecord(block.number.add(unstakingLockupBlockDelta),
        amount.add(previousPending));
    pendingWithdrawalAmount = pendingWithdrawalAmount.add(amount);

    uint256 stakedDiff = userStaked[msg.sender].sub(amount);
    currentStakedAmount = currentStakedAmount.sub(userStaked[msg.sender]);

    uint256 distributed = distribute(msg.sender);
    userStaked[msg.sender] = 0;
```

Listing 2.1: PoolStaking.sol

Impact A malicious actor can withdraw all LP tokens staked in the `PoolStaking` contract.

Suggestion Add corresponding checks to prevent the reentrancy attacks.

Feedback from the Project This issue was in unreleased code that was given to audit while still in review and testing by the team. It was found and fixed before it could be deployed. Moving away from ERC-777 will make this not possible in the future.

2.1.2 Lack of access control in airdrop contract

Severity Medium

Status Fixed* (see the *Feedback from the Project*)

Introduced by `Version 1`

Description The `BaconCoinAirdrop` contract is used to distribute airdrops of BaconCoin. It is controlled by a lock indicating whether the airdrop is allowed. However, the function for locking the airdrop is public with **NO access control**, which means that anyone is allowed to lock the airdrop.

```
function lockAirDrop() public {
    locked = true;
}
```

Listing 2.2: Airdrop.sol

Impact Anyone can lock the airdrop, and it would be unable to unlock.

Suggestion Add proper access controls.

Feedback from the Project Not a concern. This contract had served its function and was no longer needed. We have now called `lock` ourselves.

2.1.3 Potential loss in sending ERC-777 tokens

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `PoolStaking.withdraw` function will send the withdrawn LP tokens to the caller, while the LP tokens are designed to be the ERC-777 tokens. However, there is a pitfall in the OpenZeppelin's ERC-777 implementation. Specifically, if the `_send` function of the ERC-777 is used, the destination is either an EOA account, or a contract which must specify an address to implement the `tokensReceived` callback function. As a result, those destination contracts that perform lending in `PoolCore` and staking in `PoolStaking` will not be able to withdraw all their assets if they are not implemented properly.

```
function _send(
    address from,
    address to,
    uint256 amount,
    bytes memory userData,
    bytes memory operatorData,
    bool requireReceptionAck
) internal virtual {
    require(from != address(0), "ERC777: transfer from the zero address");
    require(to != address(0), "ERC777: transfer to the zero address");

    address operator = _msgSender();

    _callTokensToSend(operator, from, to, amount, userData, operatorData);

    _move(operator, from, to, amount, userData, operatorData);
```

```
    _callTokensReceived(operator, from, to, amount, userData, operatorData, requireReceptionAck
        );
}
```

Listing 2.3: openzeppelin-contracts/ERC777.sol

```
function _callTokensReceived(
    address operator,
    address from,
    address to,
    uint256 amount,
    bytes memory userData,
    bytes memory operatorData,
    bool requireReceptionAck
) private {
    address implementer = _ERC1820_REGISTRY.getInterfaceImplementer(to,
        _TOKENS_RECIPIENT_INTERFACE_HASH);
    if (implementer != address(0)) {
        IERC777Recipient(implementer).tokensReceived(operator, from, to, amount, userData,
            operatorData);
    } else if (requireReceptionAck) {
        require(!to.isContract(), "ERC777: token recipient contract has no implementer for
            ERC777TokensRecipient");
    }
}
```

Listing 2.4: openzeppelin-contracts/ERC777.sol

Impact Funds of contracts with improper implementation might be locked.

Suggestion N/A

Feedback from the Project This has been a drawback of using ERC-777 and integrating with other contracts. It provides some safety in other cases. We are already planning a change to ERC-20 that will happen shortly, but after the audit is published.

2.1.4 Potential DoS caused by insufficient gas

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `PoolStaking` contract has a complex design so that each stake (and unstake) activity may cause a specific "event" logged in the `updateEventBlockNumber` and `updateEventNewAmountStaked` state variables. The events will be continuously generated without any aggregation (e.g., one event per block), and they will be accumulated without any deletion. Therefore, each call to the `stake`, `unstake` and `distribute` functions will trigger the iteration of the historical events.

This issue could be exploited to launch the DoS attack. Specifically, if the contract receives too many `stake` and `unstake` requests (e.g., a malicious attacker calls the `stake` and `unstake` functions repeatedly), it will be expensive to invoke these functions due to the high gas consumption. Eventually all the invocations to these functions would fail because of the insufficient gas, which will lead to the DoS of this contract. As a result, the users' funds will be locked as well.

```
function stakeInternal(address wallet, uint256 amount) internal returns (bool) {
    //First handle the case where this is a first staking
    if(userStaked[wallet] != 0 || wallet == guardianAddress || wallet == daoAddress) {
        distribute(wallet);
    } else {
        userLastDistribution[wallet] = block.number;
    }

    userStaked[wallet] = userStaked[wallet].add(amount);
    currentStakedAmount = currentStakedAmount.add(amount);
    updateEventBlockNumber.push(block.number);
    updateEventNewAmountStaked.push(currentStakedAmount);
    updateEventCount = updateEventCount.add(1);

    return true;
}
```

Listing 2.5: PoolStaking.sol

Impact Users' funds could be locked in the `PoolStaking` contract due to the potential DoS problem.

Suggestion N/A

Communication with the Project

The Project: This issue would require an attacker to pay a lot of gas themselves just to DoS the project. Reducing gas costs on this function are already a top priority and will be addressed.

The Project: We propose to add a new argument to the `distribute` function: an end block for the iteration. Then, distribute would only run through the loops to get to that block. We already keep track of the last distributed block per user and start from that location on the next call. This would let people page through the `distribute` function and get past a failing transaction so at least the funds are not locked, even if it does take a few more transactions to get the funds out.

The Auditor: The fix allows pagination to the `distributeWithinBounds` function so that users can call this function multiple times to prevent gas usage problems causing the funds to be locked. However, normal usage of the `distribute` function (as in `stake` and `unstake`) still requires iterating through all events and making storage loads and comparisons.

2.1.5 Inconsistent decimal usage in `PoolStaking.distribute`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `PoolStaking.distribute` function is used to distribute BaconCoin rewards to all the stakers. This function is public accessible (i.e., it can be invoked directly), and it is called in the `stake` and `unstake` functions as well. Unfortunately, there exist several inconsistent decimal usage in the function. Specifically, in the following code snippet, at Line 240, the `tempAccruedBacon` variable has a decimal of

18; however, in Line 255, the same variable (in different branches) has a decimal of 12. This is due to an incorrect implementation of the `calcBaconBetweenEvents` function.

```
232    for (uint256 i = 0; i < updateEventCount; i++) {
233        //only accrue bacon if event is after last withdraw
234        if (updateEventBlockNumber[i] > countingBlock) {
235            blockDifference = updateEventBlockNumber[i] - countingBlock;
236
237            if(updateEventBlockNumber[i] < oneYearBlock) {
238                //calculate bacon accrued if update event is within the first year
239                //use updateEventNewAmountStaked[i-1] because that is the
240                tempAccruedBacon = blockDifference.mul(COMMUNITY_REWARD_BONUS).mul(
                       usersCurrentStake).div(updateEventNewAmountStaked[i-1]);
241            } else {
242                //calculate bacon accrued if update event is past the first year
243                if(countingBlock < oneYearBlock) {
244                    //calculate the bacon accrued at the end of the first year if overlapped with
                           first year
245                    uint256 blocksLeftInFirstYear = oneYearBlock - countingBlock;
246                    tempAccruedBacon = blocksLeftInFirstYear.mul(COMMUNITY_REWARD_BONUS).mul(
                           usersCurrentStake).div(updateEventNewAmountStaked[i-1]);
247
248                    //add the amount of bacon accrued before the first year to the running total and
                            set the block difference to start calculating from new year
249                    accruedBacon = accruedBacon.add(tempAccruedBacon);
250                    countingBlock = oneYearBlock;
251                }
252
253                //calculate the amount of Bacon accrued between events
254                uint256 baconBetweenBlocks = calcBaconBetweenEvents(countingBlock,
                       updateEventBlockNumber[i]);
255                tempAccruedBacon = baconBetweenBlocks.mul(usersCurrentStake).div(
                       updateEventNewAmountStaked[i-1]);
256            }
257
258            //as we iterate through events since last withdraw, add the bacon accrued since the
                   last event to the running total & update contingBlock
259            accruedBacon = accruedBacon.add(tempAccruedBacon);
260            countingBlock = updateEventBlockNumber[i];
261        }
262
263    }// end updateEvent for loop
```

Listing 2.6: PoolStaking.sol

Impact Inconsistent decimal usages may result in the wrong calculation of rewards.

Suggestion N/A

Feedback from the Project This code is not yet in use because we have not reached the one year block.

2.1.6 No `delegate` function implemented in BaconCoin contract

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The BaconCoin is an ERC-777 token with voting power. The general contract design of the BaconCoin follows the COMP token from the Compound project, which provides two functions, i.e., `delegate` and `delegateWithSig`, for a delegator to perform the delegation directly and indirectly. However, the BaconCoin contract does not implement the `delegate` function. Though the `delegateWithSig` function is preserved, it requires the signature of the delegator. It is suggested that BaconCoin should also provide the `delegate` function to allow the direct delegation.

Impact N/A

Suggestion N/A

Feedback from the Project We found, fixed and deployed this just before the audit started.

2.1.7 Unnecessary usage of ERC-777 standard

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Both BaconCoin and the token of `PoolCore` (i.e., the LP token) uses the ERC-777 standard. However, it seems that the ERC-777 specific features are not used. Besides, the ERC-777 token standard has a very complex callback logic, and may introduce some potential security problems. The usage of the ERC-777 also makes other projects hard to integrate with the Bacon Protocol because of the comprehensive security concerns.

Impact N/A

Suggestion Make the BaconCoin a plain ERC-20 token.

2.2 DeFi Security

2.2.1 Risk-free interest rate reduction by reentrancy

Severity Low

Status Fixed* (see the *Feedback from the Project*)

Introduced by `Version 1`

Description The BaconCoin project is a lending project where the house holders may register and stake their houses as NFTs to borrow other crypto assets. Liquidity providers (LPs) can lend assets (currently USDC) and earn interests. LPs first call the `PoolCore.lend` function to provide liquidity and receive LP tokens, which is ERC-777 enabled tokens. Borrowers call the `PoolCore.borrow` function to mortgage their home NFTs and borrow from the pool. The interest rate will be calculated and fixed when invoking the `PoolCore.borrow` function.

There exists a sequence of calls where malicious borrowers can significantly lower the borrow interest rate in a risk-free manner, as described in the following:

1. Suppose a borrower named A possesses a home NFT.
2. A first borrows the flash loan in USDC, then calls `PoolCore.lend` to provide some liquidity.
3. A then calls the `PoolCore.redeem` function to **burn** the LP tokens. Because the LP tokens are ERC-777 tokens, a callback function specified by A will be invoked **before** the `totalSupply` of LP tokens and `poolLent` state are reduced. It is worth noting that in Line 220, the redeemed token price is calculated before the callback, so it will not be affected by whatever he does in the callback function.

```
212    function redeem(
213        uint256 amount
214    ) public {
215        //check to see if sender has enough hc_pool to redeem
216        require(balanceOf(msg.sender) >= amount);
217
218        //check to make sure there is liquidity available in the pool to withdraw
219        uint256 tokenPrice = poolLent.mul(1000000).div(super.totalSupply());
220        uint256 erc20ValueOfTokens = amount.mul(tokenPrice).div(1000000);
221        require(erc20ValueOfTokens <= (poolLent - poolBorrowed));
222
223        //burn hcPool first
224        super._burn(msg.sender, amount, "", "");
225        poolLent = poolLent.sub(erc20ValueOfTokens);
226        IERC20Upgradeable(ERCAddress).transfer(msg.sender, erc20ValueOfTokens);
227    }
```

Listing 2.7: PoolCore.sol

4. During the callback, the `poolLent` variable is not updated yet and according to the interest rate fomula shown below, the calculated interest rate will be lower because the pool holds a larger amount of liquidity at this point. A can borrow at a much lower interest rate by calling the `PoolCore.borrow` function.

```
        int256 newUtilizationRatio = int256(poolBorrowed).add(int256(amount)).mul(100000000).
            div(int256(poolLent));
```

Listing 2.8: PoolUtils.sol

The difference between this malicious action and simply calling `lend`, `borrow` and `redeem` in the `PoolCore` contract sequentially with the flash loan is that the reentrancy in the callback function will make the process risk-free.

Impact Borrowers can significantly lower the interest rate with no risk and cost.

Suggestion N/A

Feedback from the Project This one is not exploitable at all because we handle all the home NFTs through the servicer wallets. The borrowers never take control of them.

2.2.2 Incorrect interest rate divisor in `PoolCore.getLoanAccruedInterest`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `PoolCore.getLoanAccruedInterest` function is used to calculate the accumulated interest of a specific loan. In this function, the `interestPerSecond` variable is calculated by using the divisor 31622400 (seconds), which is 366 days. It does not follow the common financial convention to calculate the interests (i.e., 1 year is standardized as 360 days).

```
function getLoanAccruedInterest(uint256 loanId) public view returns (uint256) {
    Loan memory loan = loans[loanId];
    uint256 secondsSincePayment = block.timestamp.sub(loan.timeLastPayment);

    uint256 interestPerSecond = loan.principal.mul(loan.interestRate).div(31622400);
    uint256 interestAccrued = interestPerSecond.mul(secondsSincePayment).div(100000000);
    return interestAccrued.add(loan.interestAccrued);
}
```

Listing 2.9: PoolCore.sol

Impact Interest rates may be incorrectly calculated.

Suggestion N/A

Feedback from the Project As long as its consistent, this is not a problem. Original implementation overlooked the standard mortgage practice. We will update to calculate interest on a 360 day year in line with standard mortgage practice.

2.2.3 No liquidation logic provided

Severity Medium

Status Acknowledged

Introduced by `Version 1`

Description Generally, the users of a lending project like BaconCoin may register their houses off-chain, and receive NFTs that are issued on-chain as equity representations. Users can then borrow crypto assets after staking the home NFTs. Under some extreme conditions, the house might be seized off-chain, and the entire on-chain system would be in bad status. Though the probability can be very low, the off-chain liquidation case should be considered. In summary, as a lending project, if no liquidation logic is provided, which suggests that in the case that the house is liquidated off-chain, there is no way to reflect the off-chain bad status on-chain.

Impact Off-chain liquidation states are unable to be reflected on-chain.

Suggestion N/A

Feedback from the Project The liquidation logic is currently embedded in the `repay` function. Since anyone can repay a loan, the servicer can pay it off after it has been liquidated. Our original roadmap includes a DAO vote to accept paying off a loan for less than the full amount in the case the collateral is insufficient and making up the shortfall using governance tokens.

2.2.4 Potential centrality problem

Severity Medium

Status Confirmed

Introduced by Version 1

Description When a user registers their houses off-chain, an on-chain NFT will be minted. This is done by calling the PropTokens.mintPropToken function. This function has an access control mechanism so that only approved servicers can call it. In such a case, the system is subject to single-point centrality problem so that if the private key of this servicer address is leaked, a malicious attacker might mint arbitrary NFTs and drain all the pools by simply borrowing with fake NFTs.

```
function mintPropToken(
    address to,
    uint256 lienValue,
    uint256[] memory seniorLienValues,
    uint256 propValue,
    string memory propAddress,
    string memory propPhotoURI
    ) public {
    //require servicer is calling
    require(isApprovedServicer(msg.sender));
```

Listing 2.10: PropTokens.sol

Impact N/A

Suggestion N/A

Feedback from the Project This is one point of centrality. We acknowledge the potential risk and use increased operational security around it to offset the concern. This address is controlled by a multi-sig wallet that requires 3 of 5 signatures and whose holders are geographically distributed.

2.2.5 Potential logic problem in PoolStaking.withdraw

Severity Low

Status Acknowledged

Introduced by Version 1

Description The PoolStaking.withdraw function treats the first address in the poolAddresses array as the pool token in this contract. Although the code logic is correct in the current implementation, it might be problematic in the future. It should be refactored into a single address to eliminate the ambiguity.

```
function withdraw(uint256 amount) public returns (uint256) {
    // Make sure that they have enough ready to withdraw
    UnstakeRecord memory userPending = userToUnstake[msg.sender];
    require(userPending.amount >= amount, "not enough pending withdraw");
    require(block.number > userPending.endBlock, "unstake still locked");

    uint256 pendingDiff = userPending.amount.sub(amount);
    userToUnstake[msg.sender].amount = pendingDiff;
    pendingWithdrawalAmount = pendingWithdrawalAmount.sub(amount);

    //finally transfer out amount
    IERC777Upgradeable(poolAddresses[0]).send(msg.sender, amount, "");

```

```
        return 0;
    }
```

Listing 2.11: PoolStaking.sol

Impact N/A

Suggestion Fix the ambiguous logic.

2.2.6 Potential incorrect logic in `PoolStaking.distribute`

Severity Low

Status Fixed

Introduced by `Version 1`

Description In the `PoolStaking.distribute` function, for some special addresses (i.e., the `daoAddress` and `guardianAddress`), the initial `userLastDistribution` would be zero (Line 212). Therefore, the variable `blockDifference` calculated in the code snippet below (Line 218 and Line 222) would be extremely large.

```
212    uint256 countingBlock = userLastDistribution[wallet];
213
214    uint256 blockDifference = 0;
215    uint256 tempAccruedBacon = 0;
216
217    if(wallet == daoAddress) {
218        blockDifference = block.number - countingBlock;
219        tempAccruedBacon = blockDifference.mul(DAO_REWARD);
220        accruedBacon += tempAccruedBacon;
221    } else if (wallet == guardianAddress) {
222        blockDifference = block.number - countingBlock;
223        accruedBacon = blockDifference.mul(GUARDIAN_REWARD);
224        accruedBacon += tempAccruedBacon;
225    } else if (countingBlock < stakeAfterBlock) {
226        countingBlock = stakeAfterBlock;
227    }
```

Listing 2.12: PoolStaking.sol

Impact N/A

Suggestion Fix the incorrect logic.

Communication with the Project

The Developer: Pretty sure this isn't a problem. We have already distributed from both addresses and received the correct amounts. Can you explain where you see the wrong initial value?

The Auditor: After reviewing older versions of the `PoolStaking` contract, we have noticed the following initialization procedure. Because newer versions are deployed as logic contracts, this issue is considered fixed currently. However if the new contract is deployed independently, the issue still exists.

```
    userLastDistribution[guardianAddress] = startingBlock;
    userLastDistribution[daoAddress] = startingBlock;
```

Listing 2.13: PoolStaking0.sol

2.3 Additional Recommendation

2.3.1 Refactor loops to mappings for gas optimization

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `isApprovedPool` function of the `PoolStaking` contract (and the `isApprovedServicer` of the `PoolCore` contract), the comparison of the address parameter and a set of allowed addresses can be simplified to a mapping to save gas.

```
function isApprovedPool(address _address) internal view returns (bool) {
    bool isApproved = false;

    for (uint i = 0; i < poolAddresses.length; i++) {
        if(_address == poolAddresses[i]) {
            isApproved = true;
        }
    }

    return isApproved;
}
```

Listing 2.14: PoolStaking.sol

Impact N/A

Suggestion Refactor loops in the above functions into mappings to save gas.

2.3.2 Remove unused variables and functions

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `PoolCore` contract, there are several state variables and functions that are not used, including `servicerAddresses`, `setApprovedAddresses`, `isApprovedServicer`, and `userLoans`.

Impact N/A

Suggestion Remove the unused variables and functions.

2.3.3 Fix wrong variable used in `PoolStaking.distribute`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `PoolStaking.distribute` function is used to distribute the BaconCoin reward to the stakers. In Line 223 of this contract, there is a misuse of the `tempAccruedBacon` and `accruedBacon` variables. It will not cause any logical errors, but should be fixed to eliminate the ambiguity.

```
217 if(wallet == daoAddress) {
218     blockDifference = block.number - countingBlock;
219     tempAccruedBacon = blockDifference.mul(DAO_REWARD);
```

```
        accruedBacon += tempAccruedBacon;
    } else if (wallet == guardianAddress) {
        blockDifference = block.number - countingBlock;
        accruedBacon = blockDifference.mul(GUARDIAN_REWARD);
        accruedBacon += tempAccruedBacon;
    } else if (countingBlock < stakeAfterBlock) {
        countingBlock = stakeAfterBlock;
    }
```

Listing 2.15: PoolStaking.sol

Impact N/A

Suggestion Fix the ambiguous logic.

2.3.4 Remove redundant logic in `PoolStaking.distribute`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the following code snippet, the conditional checks in Line 269 and Line 271 are duplicate.

```
269    if(countingBlock != block.number && countingBlock < block.number) {
270        //case where still within first year
271        if(countingBlock < oneYearBlock && block.number < oneYearBlock) {
272            //calculate accrued between last updateEvent and now
273            blockDifference = block.number - countingBlock;
274            tempAccruedBacon = blockDifference.mul(COMMUNITY_REWARD_BONUS).mul(usersCurrentStake).
                   div(updateEventNewAmountStaked[updateEventCount-1]);
275        } else {
```

Listing 2.16: PoolStaking.sol

Impact N/A

Suggestion Remove the duplicate conditional checks.