SMART CONTRACT AUDIT REPORT

for

BNPL Pay

Prepared By: Xiaomi Huang

Hangzhou, China

May 16, 2022

Document Properties

Client	BNPL
Title	Smart Contract Audit Report
Target	BNPL Pay
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 16, 2022	Jing Wang	Final Release
1.0-rc	March 16, 2022	Jing Wang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the BNPL Pay design document and related smart contract source code, 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 branch of BNPL Pay 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 BNPL Pay

BNPL Pay is a decentralized finance protocol which aims to create a unique uncollateralized lending platform. The protocol allows users to borrow funds through its system of distributed P2P lenders run natively on the Ethereum blockchain. There are four key stakeholders within the BNPL Pay ecosystem including Banking Nodes, Lenders, Borrowers, and Token Stakers.

The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of BNPL Pay

Item	Description
Name	BNPL
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 16, 2022

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

- https://github.com/BNPLPayTech/BNPL.git (57f2d99)

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

- https://github.com/BNPLPayTech/BNPL.git (c01128a)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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

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

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) [11], 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 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 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 1.4: 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 `BNPL Pay` implementation. 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 logic, 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	3	■■■
Low	4	■■■■
Informational	0	
Total	7	

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 3 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key BNPL Pay Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Logic of Calculation For principalLost Amount	Business Logic	Fixed
PVE-002	Medium	Proper Handling Of totalUnbondingShares Calculation	Business Logic	Fixed
PVE-003	Low	Reentrancy Risk in BankingNode	Business Logic	Partially Fixed
PVE-004	Medium	Possible Costly LPs From Improper BankingNode Initialization	Time and State	Fixed
PVE-005	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-006	Low	Incompatibility with Deflationary Tokens	Business Logic	Confirmed
PVE-007	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Business Logic	Confirmed

Beside the identified issues, 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 Improved Logic of Calculation For principalLost Amount

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: BankingNode
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The BNPL Pay protocol allows the user to create, and operate a pool of liquidity that is delegated to them from lenders. When the capital loss is incurred from loan defaults, the slashing occurs. The percentage slashing penalty will be equivalent to the size of the default as a percentage of the total pool capital. While examining this part of logic, we notice an issue in current implementation. To elaborate, we show below the related routines.

```
function slashLoan(uint256 loanId, uint256 minOut)
    external
    ensurePrincipalRemaining(loanId)
{
    //Step 1. load loan as local variable
    Loan storage loan = idToLoan[loanId];
    ...
    //Step 4. calculate the amount to be slashed
    uint256 principalLost = loan.principalRemaining;
    //Check if there was a full recovery for the loan, if so
    if (baseTokenOut >= principalLost) {
        ...
    }
    //slash loan only if losses are greater than recovered
    else {
        //safe div: principal > 0 => totalassetvalue > 0
        uint256 slashPercent = (1e12 * principalLost) /
            getTotalAssetValue();
        uint256 unbondingSlash = (unbondingAmount * slashPercent) / 1e12;
```

```
            uint256 stakingSlash = (getStakedBNPL() * slashPercent) / 1e12;
            //Step 5. deduct slashed from respective balances
            accountsReceiveable -= principalLost;
            slashingBalance += unbondingSlash + stakingSlash;
            unbondingAmount -= unbondingSlash;
        }
        ...
    }
```

Listing 3.1: `BankingNode::slashLoan()`

The `slashLoan()` routine implements a rather straightforward logic in allowing the users to declare a loan defaulted and slash the loan. It comes to our attention that the calculation of `principalLost` is using `(1e12 * principalLost)/ getTotalAssetValue()`. This logic makes an implicit assumption of `principalLost` is the total loss while this value should equal to `principalLost - baseTokenOut`.

Recommendation Revise the above `slashLoan` routine to properly compute the value of `principalLost`.

Status This issue has been fixed in the commit: `1f791a6`.

3.2 Proper Handling Of totalUnbondingShares Calculation

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BankingNode`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `BNPL Pay` protocol allows the user to stake `BNPL` tokens into the `Banking Nodes`. `Stakers` will receive a share of all revenues generated by the underlying pool and be subject to the same slashing penalties as those incurred by the node. To prevent gaming of the system via hopping between pools prior to revenue accrual, when tokens are withdrawn they will be subject to a 7 day unstaking period during which time no rewards will be accrued but slashing penalties can still be incurred. While examining the related implementation, we notice there is a logic error in the `unstake()` routine. To elaborate, we show below the related code snippet.

```
    function unstake() external {
        uint256 _userAmount = unbondingShares[msg.sender];
        if (_userAmount == 0) {
            revert ZeroInput();
        }
        //assuming 13s block, 46523 blocks for 1 week
```

```
        if (block.number < unbondBlock[msg.sender] + 46523) {
            revert LoanStillUnbonding();
        }
        uint256 _unbondingAmount = unbondingAmount;
        uint256 _totalUnbondingShares = totalUnbondingShares;
        address _bnpl = BNPL;
        //safe div: if user amount > 0, then totalUnbondingShares always > 0
        uint256 _what = (_userAmount * _unbondingAmount) /
            _totalUnbondingShares;
        //transfer the tokens to user
        TransferHelper.safeTransfer(_bnpl, msg.sender, _what);
        //update the balances
        unbondingShares[msg.sender] = 0;
        unbondingAmount -= _what;

        emit bnplWithdrawn(msg.sender, _what);
    }
```

Listing 3.2: `BankingNode::unstake()`

The `unstake()` routine (see the code snippet above) is provided to withdraw `BNPL` from a bond once unstaking period ends. It comes to our attention that the balance calculation of `totalUnbondingShares` is not counted the amount withdrawn by the `Stakers` into it. Hence, the later `Stakers` is subjected to a lower withdrawn amount as the `unbondingAmount` is deducted as normal while the `totalUnbondingShares` is not.

Recommendation Correct the above calculation of `totalUnbondingShares`.

Status The issue has been fixed by this commit: `1f791a6`.

3.3 Reentrancy Risk in BankingNode

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BankingNode`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `BankingNode` as an example, the `unstake()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 631) starts before effecting the update on the internal state (lines 633-634), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
615     function unstake() external {
616         uint256 _userAmount = unbondingShares[msg.sender];
617         if (_userAmount == 0) {
618             revert ZeroInput();
619         }
620         //assuming 13s block, 46523 blocks for 1 week
621         if (block.number < unbondBlock[msg.sender] + 46523) {
622             revert LoanStillUnbonding();
623         }
624         uint256 _unbondingAmount = unbondingAmount;
625         uint256 _totalUnbondingShares = totalUnbondingShares;
626         address _bnpl = BNPL;
627         //safe div: if user amount > 0, then totalUnbondingShares always > 0
628         uint256 _what = (_userAmount * _unbondingAmount) /
629             _totalUnbondingShares;
630         //transfer the tokens to user
631         TransferHelper.safeTransfer(_bnpl, msg.sender, _what);
632         //update the balances
633         unbondingShares[msg.sender] = 0;
634         unbondingAmount -= _what;
635 
636         emit bnplWithdrawn(msg.sender, _what);
637     }
```

Listing 3.3: `BankingNode::unstake()`

Note that other routines including `withdrawCollateral()`, `stake()`, `slashLoan()`, `sellSlashed()`, `makeLoanPayment()`, `requestLoan()` and `repayEarly()` from the same contract share the same issue.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been partially fixed by this commit: `3e1f4d9`.

3.4 Possible Costly LPs From Improper BankingNode Initialization

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `BankingNode`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [2]

Description

The `BankingNode` contract allows the lenders to deposit their funds to receive `bUSD` token as shares. The lenders will get their pro-rata share based on their deposited amount. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. This `deposit()` routine is used for participating lenders to deposit the supported asset (e.g., `baseToken`) and get respective shares in return. The issue occurs when the `BankingNode` contract is being initialized under the assumption that the current contract is empty.

```
    function deposit(uint256 _amount)
        external
        ensureNodeActive
        nonZeroInput(_amount)
    {
        //check the decimals of the baseTokens
        address _baseToken = baseToken;
        uint256 decimalAdjust = 1;
        uint256 tokenDecimals = ERC20(_baseToken).decimals();
        if (tokenDecimals != 18) {
            decimalAdjust = 10**(18 - tokenDecimals);
        }
        //get the amount of tokens to mint
        uint256 what = _amount * decimalAdjust;
        if (totalSupply() != 0) {
            //no need to decimal adjust here as total asset value adjusts
            //unable to deposit if getTotalAssetValue() == 0 and totalSupply() != 0, but this
            //should never occur as defaults will get slashed for some base token recovery
            what = (_amount * totalSupply()) / getTotalAssetValue();
        }
        //transfer tokens from the user and mint
        TransferHelper.safeTransferFrom(
            _baseToken,
            msg.sender,
```

```
500                 address(this),
501                 _amount
502             );
503             _mint(msg.sender, what);
504
505             _depositToLendingPool(_baseToken, _amount);
506
507             emit baseTokenDeposit(msg.sender, _amount);
508         }
```

Listing 3.4: `BankingNode::deposit()`

Specifically, when the contract is being initialized, the share value directly takes the value of `_amount` (line 489), supposing the `decimalAdjust` is 1, which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = 1 WEI`. With that, the actor can further deposit a huge amount of `baseToken` into the `lendingpool` contract on behalf of the `BankingNode` with the goal of making the share extremely expensive.

An extremely expensive share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of share calculation to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status The issue has been fixed by this commit: `607bdce`.

3.5 Accommodation of approve() Idiosyncrasies

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BankingNode`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [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 analyze 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 3.5: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `BankingNode::_depositToLendingPool()` routine as an example. This routine is designed to approve the `lendingpool` contract to deposit `tokenIn` into `aToken`. To accommodate the specific idiosyncrasy, for each `safeApprove()` (line 863), there is a need to `safeApprove()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function _depositToLendingPool(address tokenIn, uint256 amountIn) private {
        TransferHelper.safeApprove(
            tokenIn,
            address(_getLendingPool()),
            amountIn
        );
        _getLendingPool().deposit(tokenIn, amountIn, address(this), 0);
    }
```

Listing 3.6: `BankingNode::_depositToLendingPool()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `safeApprove()`.

Status The issue has been fixed by this commit: `db22368`.

3.6 Incompatibility with Deflationary Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BNPLRewardsController`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `BNPL Pay` protocol, the `BNPLRewardsController` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransferFrom()` routine to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    /**
     * Deposit LP tokens from the user
     */
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        updatePool(_pid);

        uint256 pending = ((user.amount * pool.accBnplPerShare) / 1e12) -
```

```
            user.rewardDebt;

        user.amount += _amount;
        user.rewardDebt = (user.amount * pool.accBnplPerShare) / 1e12;

        if (pending > 0) {
            safeBnplTransfer(msg.sender, pending);
        }
        TransferHelper.safeTransferFrom(
            address(pool.lpToken),
            msg.sender,
            address(this),
            _amount
        );

        emit Deposit(msg.sender, _pid, _amount);
    }

    /**
     * Withdraw LP tokens from the user
     */
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        if (_amount > user.amount) {
            revert InsufficientUserBalance(user.amount);
        }

        updatePool(_pid);

        uint256 pending = ((user.amount * pool.accBnplPerShare) / 1e12) -
            user.rewardDebt;

        user.amount -= _amount;
        user.rewardDebt = (user.amount * pool.accBnplPerShare) / 1e12;

        if (pending > 0) {
            safeBnplTransfer(msg.sender, pending);
        }
        TransferHelper.safeTransfer(address(pool.lpToken), msg.sender, _amount);

        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.7: `BNPLRewardsController::deposit()`and `BNPLRewardsController::withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accBnplPerShare` via dividing `bnplReward` by `lpSupply`, where the `lpSupply` is derived from `balanceOf(address(this))` (line 130). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may give a big `pool.accBnplPerShare` as the final result, which dramatically inflates the pool's reward.

```
122     /**
123      * Update reward variables for a pool given pool to be up-to-date
124      */
125     function updatePool(uint256 _pid) public {
126         PoolInfo storage pool = poolInfo[_pid];
127         if (block.timestamp <= pool.lastRewardTime) {
128             return;
129         }
130         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
131         if (lpSupply == 0) {
132             pool.lastRewardTime == block.timestamp;
133             return;
134         }
135         uint256 multiplier = getMultiplier(
136             pool.lastRewardTime,
137             block.timestamp
138         );
139         uint256 bnplReward = (multiplier * bnplPerSecond * pool.allocPoint) /
140             totalAllocPoint;
141 
142         //instead of minting, simply transfers the tokens from the owner
143         //ensure owner has approved the tokens to the contract
144 
145         address _bnpl = bnpl;
146         address _treasury = treasury;
147         TransferHelper.safeTransferFrom(
148             _bnpl,
149             _treasury,
150             address(this),
151             bnplReward
152         );
153 
154         pool.accBnplPerShare += (bnplReward * 1e12) / lpSupply;
155         pool.lastRewardTime = block.timestamp;
156     }
```

Listing 3.8: `BNPLRewardsController::updatePool()`

One 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 `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `BNPLRewardsController` protocol for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Note another routine, i.e., `withdrawCollateral()`, from the `BankingNode` contract shares the same issue.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate.

Status This issue has been confirmed. The team clarifies they will not support deflationary tokens.

3.7 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BankingNode`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

The `BankingNode` contract has a helper routine, i.e., `collectFees()`, that is designed to convert the `baseToken` to `BNPL` for `Stakers`. It has a rather straightforward logic in calling the `safeTransfer()` to transfer the funds and calling `_swapToken()` to actually perform the intended token swap.

```
    function collectFees() external {
        //requirement check for nonzero inside of _swap
        //33% to go to operator as baseToken
        address _baseToken = baseToken;
        address _bnpl = BNPL;
        address _operator = operator;
        uint256 _operatorFees = IERC20(_baseToken).balanceOf(address(this)) / 3;
        TransferHelper.safeTransfer(_baseToken, _operator, _operatorFees);
        //remainder (67%) is traded for staking rewards
        //no need for slippage on small trade
        uint256 _stakingRewards = _swapToken(
            _baseToken,
            _bnpl,
            0,
```

```
            IERC20(_baseToken).balanceOf(address(this))
        );
        emit feesCollected(_operatorFees, _stakingRewards);
    }
```

Listing 3.9: `BankingNode::collectFees()`

To elaborate, we show above the `collectFees()` routine. We notice the actual swap operation `_swapToken()` essentially do not specify any restriction (with `minOut=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller return for this round of operation.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense. Note the same issue also exists on the another routine in the `BankingNode` contract.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the team. And the team clarifies that since these fees will be relatively small, it is very unlikely to cause much losses from sandwich attacks.

4 Conclusion

In this audit, we have analyzed the `BNPL Pay` design and implementation. `BNPL Pay` is a decentralized finance protocol which aims to create a unique uncollateralized lending platform. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

[14] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[15] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.