QuillAudits

Audit Report
December, 2021

For

Contents

Scope of Audit	01
Check Vulnerabilities	02
Techniques and Methods	03
Issue Categories	04
Number of security issues per severity.	04
Functional Testing Results	05
Issues Found – Code Review / Manual Testing	06
High Severity Issues	06
PiStakingVault2	06
Quantity and Valuation mismatch	06
PiStakingRewards1 and PiStakingRewards2	07
Incorrect Multiplier Factor calculation	07
PiVaultStratLP and PiVaultStratToken	08
Incorrect address check	08
No Slippage Factor may lead to Sandwich Attacks	08
PiStakingVault1	09
Users may deposit multiple ERC721 NFTs	09
Incorrect calculation of Pi token amount to swap	10
Incorrect Flow of NFTs	11
Incorrect time lock Implementation	12
Medium Severity Issues	13
PiVaultStratLP and PiVaultStratToken	13
Incorrect token check	13

Contents

Missing Initialization Checks 14

Not resetting the migrateVault parameters 15

Incorrect token check 16

PiStakingVault1 17

Incorrect order or checks 17

Missing Important checks in the add function. 18

Owner can Modify timelock anytime after initialization 18

Low Severity Issues 19

VaultRewardDistributor 19

Missing Zero Address Checks 19

Array Length Mismatch may result into INVALID opcode 19

PiStakingVault2 20

Missing Zero Address Checks 20

Owner can remove rewardsToken token’s liquidity 20

Incorrect decimal assumptions 20

PiVaultStratLP and PiVaultStratToken 21

function earn withdraws the earned rewards 21

Missing correct checks 22

Unused tokens/state variables 23

PiStakingVault1 23

Unsafe Arithmetic Calculations 23

No Margin for Deadline may revert the swap transaction 24

Inefficient operations to remove multiple indexes 24

Contents

Possibly incorrect conditional logic 25

Informational 26

Public functions that are never called by the contract 26

VaultRewardDistributor 26

Default visibility for state variable 26

Missing Events for critical parameters 26

Unused function parameters 27

PiStakingVault2 27

[#L77] rewardsDuration is initialized 27

As the contract considers 18 decimals for logical operations 27

Contract supports staking of any token ID 27

PiStakingRewards1 28

Incorrect Decimal Assumptions 28

function addNFTasMultiplier sets a boostPercentNFT 29

function boostByNFT checks 29

Unused State Variables 30

Incorrect require check 30

PiStakingRewards2 31

Incorrect Decimal Assumptions 31

[#L265] function addERC1155IndexAsMultiplier 32

[#L331] function boostByERC1155NFT 32

Unused State Variables 33

Incorrect require check 33

Contents

PiStakingVault1 34

Modifier nonDuplicated 34

Make sure the max_slots for a pool 34

Consider reviewing operational logic 34

Use of Strict Inequalities 35

Undocumented/Unused parameters of a Pool 36

Default Visibility for state Variable 36

PiVaultStratLP and PiVaultStratToken 37

Missing Important checks 37

Commented code 37

Closing Summary 38

Overview

Pi BSC Vaults by Pi Protocol

Scope of the Audit

The scope of this audit was to analyze **Pi Protocol smart contract**'s codebase for quality, security, and correctness.

Pi Protocol Contracts
Commit: defe9a3673d4765da81280538843156264b4cb5c
Branch: master
Fixed In: 7ded759faa531ccadb836890d53bc0ee5190062b

VaultRewardDistributor.sol
PiStakingVault1.sol
PiStakingVault2.sol
PiVaultStratToken.sol
PiVaultStratLP.sol
PiStakingRewards.sol
PiStakingRewards2.sol

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return
- boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Mythril, Slither, SmartCheck, Surya, Solhint.

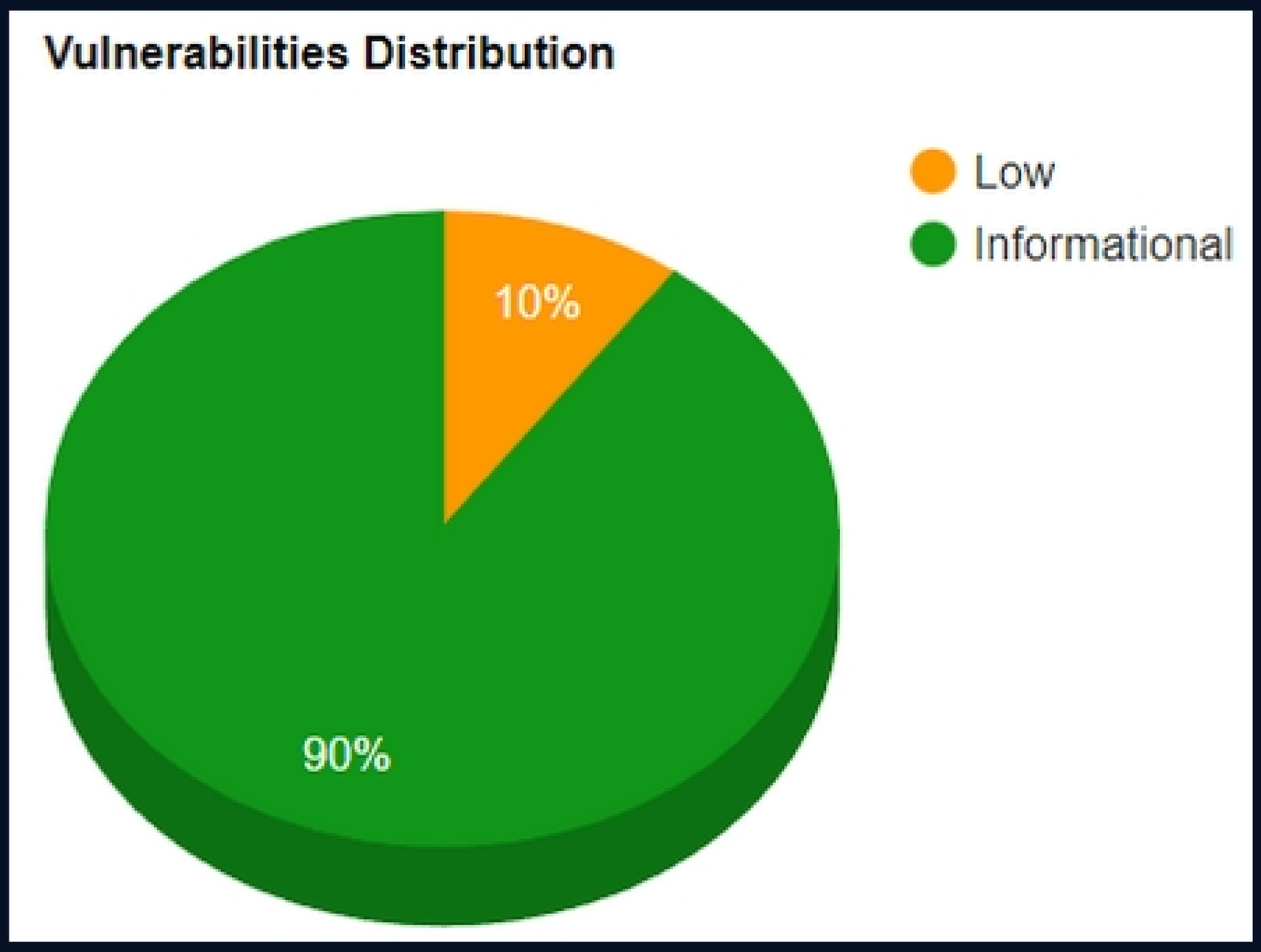
Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	0	1	9
Closed	8	7	11	18



Functional Testing Results

The test case report is available at:

PI BSC TestCase Report

Note: UniswapV2Library.**getAmountsOut** and functions depending on it didn’t work, while conducting tests on the testnet.

Issues Found

High severity issues

PiStakingVault2

- **Quantity and Valuation mismatch may lead to unoperational withdrawal or locking of funds**

Function **stake**, allows users to stake their tokens, but considers a valuation of 1000 times while withdrawing.

```
132     function withdraw(uint256 tokenId, uint256 quantity) public  nonReentrant updateReward(_msgSender()) {
133
134         // require(tokenId == 1, "Token id should be 1");
135         uint256 amount = quantity.mul(1000);
136         require(amount > 0, "Cannot withdraw 0");
137         _tokenBalances[_msgSender()][tokenId]=_tokenBalances[_msgSender()][tokenId].sub(quantity);
138
139         _totalSupply = _totalSupply.sub(amount);
140         _balances[_msgSender()] = _balances[_msgSender()].sub(amount);
141         stakingToken.safeTransferFrom(address(this),_msgSender(), tokenId, quantity, "0x");
142         emit Withdrawn(_msgSender(), tokenId, quantity, amount);
143     }
```

As a consequence, users' tokens may get locked inside the contract, or simple withdrawal may not be possible.

Some Possible Scenarios:
- A user stakes 500 tokens, but as withdrawal considers a valuation of 1000 times the amount of tokens to widraw, user can not withdraw any of its tokens
- A user stakes 5000 tokens, so the maximum withdrawable tokens will be 5 for the user, hence remaining 4995 tokens will get locked forever.

Status: Fixed

PiStakingRewards1 and PiStakingRewards2

- **Incorrect Multiplier Factor calculation may lead to inappropriate rewards generation**

  Function **getMultiplyingFactor** and **getMultiplyingFactorWei** calculate total multiplier factor for a user, by adding **multiplierFactor** from staking multiplier token and **multiplierFactorNFT** from NFTs. But under a special case, i.e., if the total multiplier is less than 1(considering 1e18 = 1), it will be added by 1, which may produce inappropriate results in many cases.

```
151     function getMultiplyingFactor(address account) public view returns (uint256) {
152         if (multiplierFactor[account] == 0 && multiplierFactorNFT[account] == 0) {
153             return 1000000;
154         }
155         uint256 MFwei = multiplierFactor[account].add(multiplierFactorNFT[account]);
156         if(MFwei<1e18){
157             MFwei = MFwei.add(1e18);
158         }
159         return MFwei.div(1e12);
160     }
161
162
163     function getMultiplyingFactorWei(address account) public view returns (uint256) {
164         if (multiplierFactor[account] == 0 && multiplierFactorNFT[account] == 0) {
165             return 1e18;
166         }
167         uint256 MFWei = multiplierFactor[account].add(multiplierFactorNFT[account]);
168         if(MFWei<1e18){
169             MFWei = MFWei.add(1e18);
170         }
171         return MFWei;
172     }
```

Some Possible Scenarios:
Let's assume, there are three NFT contracts with boost percentages 50%, 50%, 10%, and there are two users A and B, who have not staked any multiplier token meaning **multiplierFactor** for the users is 0

- User A, stakes one NFT with boost percentage as 50%, i.e 0.5, as it is less than 1, it will be added to 1 and the resulting factor will be 1.5. On the other hand User B stakes all three NFTs, totaling a factor of 1.1(0.5 +0.5+0.1). Even though user B has staked three NFTs of boost worth 110% as compared to user A which is just 50%, still user A will get more rewards because its multiplier factor is 1.5 greater than B's which is 1.1
- User A stakes two NFTs of boost worth 50% each, making the total multiplier factor as 1(0.5+0.5). So a multiplier factor of 1 will not boost the rewards anyhow for this user, making 100% boost from NFTs meaningless.

Status: Fixed

PiVaultStratLP and PiVaultStratToken

- **Incorrect address check may cause migration impossible to happen**

  [#L435] function **migratePCSVault** sets the migrate vault parameters for new PCSVault. The function checks that the **migrateVault.newPCSVaultAddress** should not be a zero address, which cannot be satisfied as **newPCSVaultAddress** doesn't hold any address yet, hence a zero address.

```
435     function migratePCSVault(uint256 _PCSPoolId, address _PCSVaultAddress) public {
436         require(msg.sender == govAddress, "!gov");
437         require(migrateVault.newPCSVaultAddress!=address(0),"Vault address cannot be zero");
438         migrateVault.newPCSPoolID= _PCSPoolId;
439         migrateVault.newPCSVaultAddress=_PCSVaultAddress;
440         migrateVault.timeMigration = block.timestamp+ waitMigTransferTime;
441     }
```

  As a consequence, migration may never happen.

  Recommendation: Consider reviewing and verifying the operational and business logic

  Status: Fixed

- **No Slippage Factor may lead to Sandwich Attacks**

  The contract doesn't incorporate any slippage factor for the swapping of tokens, and expects min output amount of tokens as 0 from the swaps. As a consequence of which, malicious actors may observe the pending transactions, and take complete advantage of it by performing sandwich attacks.

  Reference: Defcon29: Preventing Sandwich Attacks
  Recommendation: Consider adding slippage factor to avoid risks of sandwich attacks

  Status: Fixed

PiStakingVault1

- Users may deposit multiple ERC721 NFTs with least native deposit fee possible for a pool

```
213     function deposit(uint256 _pid, uint256 totalNFTs, uint256[] memory _erc721tokenIds) public nonReentrant {
214         require(totalNFTs>0,"amount of totalNFTs to transfer cannot be 0");
215
216         PoolInfo storage pool = poolInfo[_pid];
217         require(slots_filled[_pid]<pool.max_slots,"Max slots filled");
218
219         uint256 _amount = totalNFTs.mul(pool.depositFeeNative);
220
221         if(pool.isERC1155){
222             require(NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]<pool.max_per_user,"Max per user already done");
223
224             IERC1155(pool.nativeNFTtoken).safeTransferFrom( msg.sender, address(this), pool.ERC1155NFTid, totalNFTs, "0x");
225             // donot duplicate erc1155 token ids
```

Function **deposit** calculates fees in terms of native tokens on the amount of ERC1155 NFTs being deposited by a user with the parameter **totalNFTs**. **totalNFTs** is the number of ERC1155 tokens a user wants to deposit.

A pool can support either ERC1155 or ERC721 token deposits. For ERC721 token deposits, a user is supposed to provide ERC721 token ids with the function call, and has no use of **totalNFTs** parameter. As a result a user may pass 1 as a value for **totalNFTs** and may stake multiple NFT tokens at once, thus costing the user the lowest fees for the deposit operation.

Recommendation: Consider adding/fixing fees calculation logic

Status: Fixed

- **Incorrect calculation of Pi token amount to swap, may be exploited to drain out all the liquidity**

  Function **deposit** calculates the Pi token amount to swap for pool's wantToken for a user as

```
244
245         uint256 allocatedPi = totalNFTs.mul(pool.PiTokenAmtPerNFT);
246         // market buy
247
248         uint256 prevWantAmount = IERC20(pool.wantToken).balanceOf(address(this));
249         uint256 prevWrappedBNBAmount = IERC20(wrappedBNB).balanceOf(address(this));
250
251         // so that stock is not deep -> making copies
252         uint poolId = _pid;
253         uint256 amount = _amount;
```

  The calculation depends upon the function parameter **totalNFTs**, which is the count of ERC1155 tokens a user wants to deposit. However, in the case of ERC721 pool, the **totalNFTs** functional parameter is optional. However, a user may pass an arbitrary large number for the parameter, and swap large amounts of Pi tokens to pool's wantToken and later withdraw them all. A user may drain out all the liquidity of the contract.

  Recommendation: Consider fixing the logic to swap an appropriate amount of Pi tokens in the case of ERC721 deposits

  Status: Fixed

- Incorrect Flow of NFTs

```
373         } else // erc721
374         {
375             uint256[] memory erc721tokenIds = userInfo[_pid][msg.sender].AllNFTIds;
376             for(uint i=0;i<erc721tokenIds.length;i++){
377                 IERC721(pool.nativeNFTtoken).transferFrom(msg.sender, pool.strat, erc721tokenIds[i]);
378                 NFTIdsDeposits[_pid][msg.sender][erc721tokenIds[i]]=0;
379                 slots_filled[_pid]-=1;
380             }
381
382             uint[] storage auxArray;
383
384             for (uint i = 0; i < userInfo[_pid][msg.sender].AllNFTIds.length; i++){
385                 uint256 id= userInfo[_pid][msg.sender].AllNFTIds[i];
386                 if(NFTIdsDeposits[_pid][msg.sender][id]>0)
387                     auxArray.push(userInfo[_pid][msg.sender].AllNFTIds[i]);
388             }
389             userInfo[_pid][msg.sender].AllNFTIds = auxArray;
390         }
391
```

[#L343] function **withdrawAll** aims to transfer deposited ERC721 NFTs back to the user. However, implemented logic transfers tries to transfer user's deposited ERC721 NFT Ids again **from the user to the pool's strategy**, which may revert the operation or produce incorrect results.

Recommendation: Consider fixing the transfer of NFT tokens

Status: Fixed

- **Incorrect time lock implementation**

  The team decided to add a time lock period in commit e19...c58 while withdrawing, for every user in order to restrict them to drain out the contract's liquidity by depositing and withdrawing in a loop.

```
364  370 +   // all NFTs will be withdrawn
365  371     function withdrawAll(uint256 _pid) public nonReentrant  {
366  372
     373 +       require(lockTimeStamp[msg.sender]>=block.timestamp,"Cannot withdraw before timelock finishes");
367  374         PoolInfo storage pool = poolInfo[_pid];
368  375         uint256 _amount = userInfo[_pid][msg.sender].amount;
369  376
```

  However, the implementation and logic of the check added is incorrect. It checks for the locktime of a user, that it should be more than or ahead of the current **block.timestamp** which will be true until the locktime passes.

  Recommendation: Fix the implementation such that it should check and allow a withdrawal, only if the current **block.timestamp** has surpassed the lock time for the user.

  Status: **Fixed**

Medium severity issues

PiVaultStratLP and PiVaultStratToken

- **Incorrect token check**

  [#L445] Function **restakeAllToMigratedVault** restakes want Tokens from existing PCSVaultAddress to **migrateVault.newPCSVaultAddress**

```
445     function restakeAllToMigratedVault() public {
446         require(msg.sender == govAddress, "!gov");
447 
448         require(block.timestamp>=migrateVault.timeMigration && migrateVault.newPCSVaultAddress!=address(0), "Cannot start transfer before wait time is passed");
449         require(block.timestamp<migrateVault.timeMigration + 2 days , "Migration time expired, should call this function within 2 days");
450 
451         require(address(IPiStakingVaults(piFarmAddress).poolInfo(piVaultPoolid).wantToken)==IPCSVault(migrateVault.newPCSVaultAddress).poolInfo(PCSPoolId).lpToken,"Pool id lpt
452 
453         uint256 amount = IPCSVault(PCSVaultAddress).userInfo(PCSPoolId,address(this)).amount;
454 
455         IPCSVault(PCSVaultAddress).withdraw(PCSPoolId, amount);
456         uint256 _bal = IERC20Upgradeable(wantAddress).balanceOf(address(this));
457 
458         IERC20Upgradeable(wantAddress).safeIncreaseAllowance(migrateVault.newPCSVaultAddress, _bal);
459         IPCSVault(migrateVault.newPCSVaultAddress).deposit(migrateVault.newPCSPoolID, _bal);
460 
461         PCSPoolId=migrateVault.newPCSPoolID;
462         PCSVaultAddress=migrateVault.newPCSVaultAddress;
463 
464     }
```

The function checks, that the piFarm's wantToken at **piVaultPoolid** matches the newPCSVault's lpToken at **PCSPoolId**, where **PCSPoolId**, actually refers to existing PCSVault's pool id, instead of **migrateVault.newPCSPoolID**. As a consequence, restaking may fail.

Recommendation: Consider replacing **PCSPoolId** with **migrateVault.newPCSPoolID**

Status: **Fixed**

- **Missing Initialization Checks**

  [#L90] function **initialize** initializes the strategy with addresses of piFarm, tokens being used, PCSVault and fee receiver's address, but lacks important checks to make sure that no incorrect initialization happens.

  Some Missing checks are:

  - Zero Address Checks for address type
  - piFarm's wantToken at piVaultPoolid, should match PCSVault's lpToken at PCSPoolId
  - earnedAddress should match the cake token being sent as reward from PCSVault
  - There should be pools, existing for route path

```
128         earnedToToken0Path = [earnedAddress, token0Address];
129         earnedToToken1Path = [earnedAddress, token1Address];
130         token0ToEarnedPath = [token0Address, earnedAddress];
131         token1ToEarnedPath = [token1Address, earnedAddress];
132         PCSToEarnedPath = [PCSTokenAddress, earnedAddress];
133
```

  - Lower/Upper Bound Checks for controllerFee

  Recommendation: Consider adding the required checks to avoid risks of incorrect initializations

  Status: **Fixed [Checks recommended in Points 1, 2 and 5 have been implemented]**

- **Not resetting the migrateVault parameters will allow restaking any number of times within two days**

  [#L445] Function **restakeAllToMigratedVault** restakes want Tokens from existing PCSVaultAddress to **migrateVault.newPCSVaultAddress**, and sets the **PCSPoolId** and **PCSVaultAddress** to new PCSVault's pool id and address.

  As the function doesn't reset the **migrateVault** parameters, there are possibilities that it may be called again, which will restake the wantTokens from existing PCSVault to existing PCSVault, with no possible benifits, as both **PCSVaultAddress** and **migrateVault.newPCSVaultAddress** now points to same PCSVault.

  Recommendation: Consider resetting the migrateVault parameters.

  Status: Fixed

- **Incorrect token check in PiVaultStratToken and Inefficient Swapping**

```
235             if (earnedAddress != token0Address) {
236                 // Swap half earned to token0
237                 IUniswapV2Router02(uniRouterAddress)
238                     .swapExactTokensForTokensSupportingFeeOnTransferTokens(
239                     earnedAmt.div(2),
240                     0,
241                     earnedToWantPath,
242                     address(this),
243                     block.timestamp + 60
244                 );
245             }
```

Function earn swaps the **earned** token balance to **want** token, but checks that the earned token should not be the same as **token0Address**, instead of checking it want token.

Also, **[#L239]** function swaps only half of the available **earned** token balance instead of utilizing the complete balance, which is inefficient.

Also, **[#L236]** an incorrect comment was found. Function swaps **earned** token with **want** token, but corresponding comment points towards the swapping of **earned** token to **token0**

Recommendation: Consider checking the earned token with appropriate destination token. Also consider utilizing the complete earned token balance.

Status: Fixed

PiStakingVault1

- Incorrect order or checks may bypass maximum slots and nft deposits per user for a pool

```
212     // _erc721tokenId send if NFT token is ERC 721
213     function deposit(uint256 _pid, uint256 totalNFTs, uint256[] memory _erc721tokenIds) public nonReentrant {
214         require(totalNFTs>0,"amount of totalNFTs to transfer cannot be 0");
215
216         PoolInfo storage pool = poolInfo[_pid];
217         require(slots_filled[_pid]<pool.max_slots,"Max slots filled");
218
219         uint256 _amount = totalNFTs.mul(pool.depositFeeNative);
220
221         if(pool.isERC1155){
222             require(NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]<pool.max_per_user,"Max per user already done");
223
224             IERC1155(pool.nativeNFTtoken).safeTransferFrom( msg.sender, address(this), pool.ERC1155NFTid, totalNFTs, "0x");
225             // donot duplicate erc1155 token ids
226             if(NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]==0){
227                 userInfo[_pid][msg.sender].AllNFTIds.push(pool.ERC1155NFTid);
228             }
229             NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]+=totalNFTs;
230             slots_filled[_pid]+=totalNFTs;
231
232         } else // erc721
233         {
234             require(userInfo[_pid][msg.sender].AllNFTIds.length<pool.max_per_user,"Max per user already done");
235             for(uint i=0;i<_erc721tokenIds.length;i++){
236                 IERC721(pool.nativeNFTtoken).transferFrom(msg.sender, address(this), _erc721tokenIds[i]);
237                 userInfo[_pid][msg.sender].AllNFTIds.push(_erc721tokenIds[i]);
238                 NFTIdsDeposits[_pid][msg.sender][_erc721tokenIds[i]]=1;
239                 slots_filled[_pid]+=1;
```

The highlighted **require checks** in function **deposit** restricts the slots and nft deposits for a user to not exceed **max_slots** or **max_per_user** parameters of a pool respectively. However, the checks consider/refer to the existing values and not to the resulting values after the successful deposit operation. Hence, the checks may be bypassed at least once.

Some Possible Scenarios:
Assume **max_slots** for a pool is set as 10 i.e maximum number of slots allowed are 9, and let's say **slots_filled** for this pool, points 9, so technically, the slot is completely filled and no deposit shall happen. But, the pool will still allow a new deposit and increase the **slots_filled** value more than the maximum allowed.
Same scenario may happened with respect to **max_per_user**

Recommendation: The checks should be done after reflecting changes to the existing values

Status: Fixed

- **Missing Important checks in add function may cause imbalance or produce inappropriate results**

  Function **add** is used to add and initialize a new pool. But the function lacks important checks, which may cause imbalance in the system.

  Some of the important checks are:
  - Make sure that **token0** and **token1** addresses of wantToken or the **wantToken** itself, should not the same as **Pi** token address, otherwise, the **deposit** function may perform swapping for the same **Pi** token
  - Make sure the **wantToken** being initialized for a pool should be the same as **wantToken** of the pool's strategy.
  - Zero Address checks for address type
  - Upper/Lower bound checks for value type

  Recommendation: Consider adding the required checks.

  Status: Fixed

- **Owner can modify timelock anytime after initialization**

  An administrative function has been added in commit: 00...866, which allows the owner to change the timelock period any time after initialization, which may affect users.

```
231 +     function changeTimeLockPeriod(uint secs) external onlyOwner{
232 +         timeLockInSecs = secs;
233 +     }
234 +
```

  Possible Scenarios:
  - Owner may reduce the timelock to 0, in order to allow anyone and everyone to drain the liquidity
  - Owner may increase the timelock to a very high value, in order to disallow any user withdrawal.

  Status: Fixed

Low severity issues

VaultRewardDistributor

- Missing Zero Address Checks:

  [#L24] constructor: for **_stakingContract** address
  [#L30] setStakingContract: for **_stakingContract** address
  [#L34] addParams: for **_feeAddress** address
  [#L42] addParamsBulk: for **_feeAddresses** addresses

  Recommendation: Add require checks

  Status: **Fixed**

- Array Length Mismatch may result into INVALID opcode

```
42      function addParamsBulk(address[] memory _feeAddresses, uint256[] memory _feePercents) external ownerOnly{
43          for(uint i=0;i<_feeAddresses.length;i++){
44              transferParams.push(
45                  TransferParams(_feeAddresses[i],_feePercents[i])
46              );
47              totalParams++;
48          }
49      }
50
```

A mismatch in the number of values for arrays **_feeAddresses** and **_feePercents** may result in transaction failure with INVALID opcode, thus consuming all the gas value of transaction as execution cost.

Recommendation: Add **require** check so as to make sure the array lengths are equal, which will revert and refund the remaining gas back to sender, if arrays of unequal lengths are supplied.

Status: **Fixed**

PiStakingVault2

- Missing Zero Address Checks:

  [#L33] setRewardsDistribution: for **_rewardsDistribution** address
  [#L64] initialize: for **_rewardsDistribution**, **_rewardsToken1**, **_stakingToken** addresses

  Status: **Fixed**

- Owner can remove rewardsToken token's liquidity

```
183     function recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner {
184
185         address owner = OwnableUpgradeable.owner();
186         IERC20(tokenAddress).transfer(owner, tokenAmount);
187         emit Recovered(tokenAddress, tokenAmount);
188     }
189
```

  Function **recoverERC20** allows Owner to recover any ERC20 token from the contract, including **rewardsToken** itself.

  Recommendation: Consider adding a check to disallow recovering of **rewardsToken**.

  Status: **Fixed**

- Incorrect decimal assumptions may lead to inappropriate visualization of rewardPerToken for a user

  The contract calculates rewards **earned** and **rewardPerToken** considering **18 decimals** for **totalSupply** and **balances[account]**, but as ERC1155 doesn't have a concept of decimals, rewardPerToken may affect user experience in terms of visualization.

```
 99     function rewardPerToken() public  view returns (uint256) {
100         if (_totalSupply == 0) {
101             return rewardPerTokenStored;
102         }
103         return
104             rewardPerTokenStored.add(
105                 lastTimeRewardApplicable().sub(lastUpdateTime).mul(rewardRate).mul(1e18).div(_totalSupply)
106             );
107     }
108
109     function earned(address account) public  view returns (uint256) {
110         return _balances[account].mul(rewardPerToken().sub(userRewardPerTokenPaid[account])).div(1e18).add(rewards[account]
111     }
```

Possible Scenario:
Without decimals (assuming 1e18) for **totalSupply** and **balances[account]**: rewardPerToken will have a factor of **1e36** in its calculated values.

With decimals (assuming 1e18) for **totalSupply** and **balances[account]:** rewardPerToken will have a factor of **1e18** in its calculated values, but **totalSupply** and **balances[account]** will now be containing a factor 1e18 as well, affecting user experience.

Commit Update: 163...f8a
The logic for decimals have been updated by developer in function **stake**

```
123 +
124 +           _totalSupply = _totalSupply.add(amount.mul(1e18));
125 +           _balances[_msgSender()] = _balances[_msgSender()].add(amount.mul(1e18));
```

But there is a requirement of the same logic in **withdraw** function as well.

Recommendation: Consider reviewing and verifying the operational and business logic

Status: Fixed

PiVaultStratLP and PiVaultStratToken

- **[#L224]** function **earn** withdraws the earned rewards from PCSVault, and distribute **controllerFee** to **feeAddress**. The remaining balance is used to swap tokens to **wantToken**. It may happen that after deducting **controllerFee**, earned tokens may become 0, hence adding a check to allow swaps only for non-zero earned tokens would be better.

Recommendation: Consider adding a check to allow only non-zero swaps after distributing fees.

Status: Fixed

- **Missing correct checks and conditions for _wantAmt to be withdrawn**

  [#L191] The function **withdraw** tries to withdraw the desired **_wantAmt** from PCSVault. Logically, Strategy can not withdraw more than what it has staked, but no checks are incorporated to counter this case.

```
183     function withdraw(address userAddress, uint256 _wantAmt)
184         public
185         StakingFarmOnly
186         nonReentrant
187         returns (uint256)
188     {
189         require(_wantAmt > 0, "_wantAmt <= 0");
190
191         IPCSVault(PCSVaultAddress).withdraw(PCSPoolId, _wantAmt);
192
193         uint256 wantAmt = IERC20Upgradeable(wantAddress).balanceOf(address(this));
194         if (_wantAmt > wantAmt) {
195             _wantAmt = wantAmt;
196         }
197
198         if (wantLockedTotal < _wantAmt) {
199             _wantAmt = wantLockedTotal;
200         }
201
202         uint256 sharesRemoved = _wantAmt.mul(sharesTotal).div(wantLockedTotal);
203         if (sharesRemoved > sharesTotal) {
204             sharesRemoved = sharesTotal;
205         }
```

Possibly incorrect checks:

1. **[#L194] _wantAmt > wantAmt** : If **L191** always withdraws **want tokens** equal to **_wantAmt**, then the condition may never be true. The condition may be satisfied, if withdrawal always happens with a value less than **_wantAmt**
2. **[#L198] wantLockedTotal < _wantAmt**: If **L191** doesn't allow withdrawal of more than what has been deposited, then the condition may never be true.

If case 1 holds true, then it may affect the calculation/removal of shares. Also, it may cause cause conflict between wantLockedTotal and the actual amount of want tokens locked in PCSVault, while restaking.

Recommendation: Consider reviewing and verifying the operational and business logic

Status: Acknowledged

- **Unused tokens/state variables**

  Contract defines state variables as **token0Address** and **token1Address**. The state variables have not been used for any logical operations.

  Recommendation: Consider reviewing and verifying the operational and business logic

  Status: **Fixed**

PiStakingVault1

- **Unsafe Arithmetic Calculations**

  The contract uses Openzeppelin's **SafeMath** library for safe arithmetic calculations at most of the places. However, some unsafe calculations have been found, which are being done without SafeMath's arithmetic security checks, as a consequence an edge case scenario may result into integer overflow/underflow.

  Some of these unsafe calculations are:

```
229             NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]+=totalNFTs;
230             slots_filled[_pid]+=totalNFTs;
```

```
238                 NFTIdsDeposits[_pid][msg.sender][_erc721tokenIds[i]]=1;
239                 slots_filled[_pid]+=1;
```

```
356         if(pool.isERC1155){
357
358             slots_filled[_pid]-=NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid];
359
```

```
378                 NFTIdsDeposits[_pid][msg.sender][erc721tokenIds[i]]=0;
379                 slots_filled[_pid]-=1;
```

  Recommendation: Use SafeMath security checks for these arithmetic calculations so as to avoid the risks of overflows/underflows

  Status: **Fixed**

- **No Margin for Deadline may revert the swap transaction**

```
464         catch{
465
466             uint slippageFactor=(SafeMathUpgradeable.sub(100000,slippage)).div(1000); // 100 - slippage => will return like 98000/1000 = 98 for default
467             address[] memory path = new address[](3);
468             path[0] = _tokenAddress;
469             path[1] = wrappedBNB;
470             path[2] = address(Pi);
471             (uint256[] memory amounts) = UniswapV2Library.getAmountsOut(address(uniswapV2Factory), _amount, path);
472             uniswapV2Router.swapExactTokensForTokens(_amount, amounts[2].mul(slippageFactor).div(100), path, _to, block.timestamp);
473             delete path;
474
475         }
```

[#L431] Function **marketBuyAndTransfer** tries to swap **_tokenAddress** token to **Pi** token with a strict deadline, and does not consider a time margin considering the network congestion, as a consequence the transaction may fail under certain conditions.

Recommendation: Consider adding a time margin for deadline while swapping

Status: Fixed

- **Inefficient operations to remove multiple indexes for ERC1155 pool**

```
356         if(pool.isERC1155){
357
358             slots_filled[_pid]-=NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid];
359
360             IERC1155(pool.nativeNFTtoken).safeTransferFrom( address(this), msg.sender, pool.ERC1155NFTid, NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid], "0x");
361             // donot duplicate erc1155 token ids
362
363             uint[] storage auxArray;
364
365             for (uint i = 0; i < userInfo[_pid][msg.sender].AllNFTIds.length; i++){
366                 if(userInfo[_pid][msg.sender].AllNFTIds[i] != pool.ERC1155NFTid)
367                     auxArray.push(userInfo[_pid][msg.sender].AllNFTIds[i]);
368             }
369
370             userInfo[_pid][msg.sender].AllNFTIds = auxArray;
371             NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]=0;
372
```

[#L343] function **withdrawAll** tries to replace **AllNFTIds** array for a user with a new array containing ERC1155 Ids excluding ERC1155NFTid of the pool. However, as there will be only one ERC1155 id for an ERC1155 pool, it makes these operations inefficient.

Recommendation: Consider reviewing and verifying the operational and business logic

Status: Fixed

- **Possibly incorrect conditional logic**

  [#L343] function **withdrawAll** defines

```
343     function withdrawAll(uint256 _pid) public nonReentrant  {
344
345         PoolInfo storage pool = poolInfo[_pid];
346         uint256 _amount = userInfo[_pid][msg.sender].amount;
347
```

  And later compares **_amount** with the same value as:

```
394         uint256 amount = userInfo[_pid][msg.sender].shares.mul(wantLockedTotal).div(sharesTotal);
395
396         if(_amount>userInfo[_pid][msg.sender].amount)
397             require(amount >= _amount, "withdraw: not good");
398
```

  The implemented check is meaningless and will always fail.

  Recommendation: Consider reviewing and verifying the operational and business logic

  Status: Fixed

Informational issues

- **Public functions that are never called by the contract should be declared external to save gas.**

  Status: **Acknowledged**

- **Setter functions lack important checks, like**:
  - Zero Address Checks for address type
  - Lower/Upper bound checks for value types

  Status: Acknowledged

VaultRewardDistributor

- **[#L22]** Default visibility for state variable totalParams has been opted.

  Recommendation: Explicitly define visibility so as to avoid incorrect assumptions for access restrictions.

  Status: Fixed

- **Missing Events for critical parameters:**

  [#L30] function **setStakingContract**: No event for **_stakingContract** updates, which will make it difficult to track down the new updates.

  Recommendation: Emit an event for critical parameter changes.

  Status: Fixed

- **Unused function parameters**

  Function **transfer** contains **from** and **to address** parameters, which don't take part in any logical operations.

```
56      function transfer(address token, address from, address to, uint256 amount) external onlyStakingContract nonReentrant returns (bool){
57          IERC20(token).transferFrom(stakingContract, address(this), amount);
58          uint256 totalFeePercent=0;
59          for(uint i=0;i<transferParams.length;i++){
60               totalFeePercent = totalFeePercent.add(transferParams[i].feePercent);
61          }
62          require(totalFeePercent==100000,"totalFeePercent should be 100%");
63
64          for(uint i=0;i<transferParams.length;i++){
65              uint256 fee = transferParams[i].feePercent.mul(amount).div(100000);
66              IERC20(token).transfer(transferParams[i].feeAddress, fee);
67          }
68
69          return true;
70      }
71
```

  Recommendation: Consider emitting an event with function parameters

  Status: Fixed

- Use SafeERC20 wrapper for IERC20 operations as a safer approach to interact with unauthorized ERC20 tokens

  Status: Fixed

PiStakingVault2

- **[#L77] rewardsDuration** is initialized with **2 weeks**, whereas the comment points **1 weeks[#L48]**

  Status: Acknowledged

- As the contract considers 18 decimals for logical operations, **rewardsToken** ERC20 token is subject to 18 decimals.

  Status: Acknowledged

- Contract supports staking of any token ID

  Recommendation: Consider reviewing and verifying the operational and business logic

  Status: Fixed

PiStakingRewards1

- **Incorrect Decimal Assumptions may provide incorrect yield information**

```
190     function getRewardToken1APY() external view returns (uint256) {
191         //3153600000 = 365*24*60*60
192         if(block.timestamp>periodFinish) return 0;
193         uint256 rewardForYear = rewardRate1.mul(31536000);
194         if(_totalSupply<=1e18) return rewardForYear.div(1e10);
195         return rewardForYear.mul(1e8).div(_totalSupply); // put 6 dp
196     }
197
198
199
200     function getRewardToken1WPY() external view returns (uint256) {
201         //60480000 = 7*24*60*60
202         if(block.timestamp>periodFinish) return 0;
203         uint256 rewardForWeek = rewardRate1.mul(604800);
204         if(_totalSupply<=1e18) return rewardForWeek.div(1e10);
205         return rewardForWeek.mul(1e8).div(_totalSupply); // put 6 dp
206     }
```

Function **getRewardToken1APY** and **getRewardToken1WPY** calculate percentage yields for **1 year** and **1 week** of time respectively. The comment mentioned in the function points to put 6 decimal points, but the logic actually puts 8 decimals (assuming **stakingToken** and **rewardsToken1** to have 18 decimals).

Recommendation: Consider reviewing and verifying the operational and business logic

Status: Acknowledged

- **[#L263]** function **addNFTasMultiplier** sets a boostPercentNFT for an NFT contract, and adds it to the **erc721NFTContracts** array. It may happen that the same NFT contract is added more than once, which will make copies of the same address in the array.

```
263     function addNFTasMultiplier(address _erc721NFTContract, uint256 _boostPercent) external onlyOwner {
264
265         require(block.timestamp >= periodFinish,
266             "Cannot set NFT boosts after staking starts"
267         );
268
269         erc721NFTContracts.push(_erc721NFTContract);
270         boostPercentNFT[_erc721NFTContract] = _boostPercent.mul(1e13);
271     }
272
```

Recommendation: It may be checked whether the contract already exists or not prior to adding it into the array.

Recommendation: It may be checked whether the contract already exists or not prior to adding it into the array.

Status: **Fixed**

- **[#L322]** function **boostByNFT** checks whether the desired NFT contract is allowed for the boost or not, by iterating the **erc721NFTContracts** array.

  Recommendation: The logic may be switched to **mapping** to reduce the number of operations, checking whether a contract is allowed or not.

  Status: **Fixed**

- **Unused State Variables**

  Contract contains some state variables which never take part in any logical operations, hence are subject to be reviewed and rechecked considering the business logic.

  The variables are:
  rewardsToken2
  rewardPerToken2Stored

  rewardsToken2 never gets initialized in any logical operation, and stays as a Zero Address, thus checking tokenAddress against it is meaningless while recovering ERC20 tokens

```
function recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner {
    // Cannot recover the staking token or the rewards token
    require(
        tokenAddress != address(stakingToken) && tokenAddress != address(stakingTokenMultiplier) && tokenAddress != address(rewardsToken1) && tokenAddress != address(rewardsToken2),
        "Cannot withdraw the staking or rewards tokens"
    );
    address owner = OwnableUpgradeable.owner();
    IERC20Upgradeable(tokenAddress).safeTransfer(owner, tokenAmount);
    emit Recovered(tokenAddress, tokenAmount);
}
```

  Recommendation: Consider reviewing and verifying the operational and business logic

  Status: Fixed

- **Incorrect require check**

```
344         // CHECK already boosted by same NFT contract??
345         require(boostedByNFT[msg.sender][_erc721NFTContract]==false,"Already boosted by this NFT");
346
347         require(totalNFTsBoostedBy[msg.sender]<=erc721NFTContracts.length,"Total boosts cannot be more than MAX NFT boosts available");
348
349         multiplierFactorNFT[msg.sender]= multiplierFactorNFT[msg.sender].add(multiplyFactor);
350         IERC721(_erc721NFTContract).transferFrom(msg.sender, devFundAdd, _tokenId);
351
352         totalNFTsBoostedBy[msg.sender]=totalNFTsBoostedBy[msg.sender].add(1);
353         boostedByNFT[msg.sender][_erc721NFTContract] = true;
354
355         emit NFTMultiplier(msg.sender, _erc721NFTContract, _tokenId);
```

  Function **boostByNFT** puts restrictions on maximum number of NFT contract boosters allowed per user by making sure that it should not exceed erc721NFTContracts array length i.e, total number of ERC721 NFT contracts available.

The check considers the existing value of **totalNFTsBoostedBy** instead of considering the new value, which will be **totalNFTsBoostedBy+1** after the successful operation, thus incorrect.

Although, it is not affecting the function's operational logic here, as it allows boosting of one NFT contract only once, so technically, the maximum possible NFT boosters per user, will be the number of ERC721 NFT contracts, i.e erc721NFTContracts array length itself.

Recommendation: Even though it is not affecting the operational logic here, consider adapting the appropriate checks.

Status: **Fixed**

PiStakingRewards2

- **Incorrect Decimal Assumptions may provide incorrect yield information**

```
193     function getRewardToken1APY() external view returns (uint256) {
194         //3153600000 = 365*24*60*60
195         if(block.timestamp>periodFinish) return 0;
196         uint256 rewardForYear = rewardRate1.mul(31536000);
197         if(_totalSupply<=1e18) return rewardForYear.div(1e10);
198         return rewardForYear.mul(1e8).div(_totalSupply); // put 6 dp
199     }
200
201
202
203     function getRewardToken1WPY() external view returns (uint256) {
204         //60480000 = 7*24*60*60
205         if(block.timestamp>periodFinish) return 0;
206         uint256 rewardForWeek = rewardRate1.mul(604800);
207         if(_totalSupply<=1e18) return rewardForWeek.div(1e10);
208         return rewardForWeek.mul(1e8).div(_totalSupply); // put 6 dp
209     }
```

Function **getRewardToken1APY** and **getRewardToken1WPY** calculate percentage yields for **1 year** and **1 week** of time respectively. The comment mentioned in the function points to put 6 decimal points, but the logic actually puts 8 decimals (assuming **stakingToken** and **rewardsToken1** to have 18 decimals).

Recommendation: Consider reviewing and verifying the operational and business logic

Status: **Acknowledged**

- **[#L265]** function **addERC1155IndexAsMultiplier** sets a boostPercentNFT for an ERC1155Index, and adds it to the **Erc1155Indexes** array. It may happen that the same index is added more than once, which will make copies of the same index in the array.

```
265     function addERC1155IndexAsMultiplier(address _NFTContract, uint256 _boostPercent, uint _erc1155Index) external onlyOwner {
266
267         require(block.timestamp >= periodFinish,
268             "Cannot set NFT boosts after staking starts"
269         );
270
271         if(ERC1155AddressBooster!=address(0))
272             require(_NFTContract == ERC1155AddressBooster,"only 1 ERC1155 contract supported");
273
274         Erc1155Indexes.push(_erc1155Index);
275
276         boostPercentNFT[_erc1155Index] = _boostPercent.mul(1e13);
277
278         ERC1155AddressBooster = _NFTContract;
279
280     }
```

Status: Fixed

- **[#L331]** function **boostByERC1155NFT** checks whether the desired ERC1155 index is allowed for the boost or not, by iterating the **Erc1155Indexes** array.

Recommendation: The logic may be switched to **mapping** to reduce the number of operations, checking whether a contract is allowed or not.

Status: Fixed

- **Unused State Variables**

  Contract contains some state variables which never take part in any logical operations, hence are subject to be reviewed and rechecked considering the business logic.

  The variables are:
  rewardsToken2
  rewardPerToken2Stored

  rewardsToken2 never gets initialized in any logical operation, and stays as a Zero Address, thus checking tokenAddress against it is meaningless while recovering ERC20 tokens

```
function recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner {
    // Cannot recover the staking token or the rewards token
    require(
        tokenAddress != address(stakingToken) && tokenAddress != address(stakingTokenMultiplier) && tokenAddress != address(rewardsToken1) && tokenAddress != address(rewardsToken2),
        "Cannot withdraw the staking or rewards tokens"
    );
    address owner = OwnableUpgradeable.owner();
    IERC20Upgradeable(tokenAddress).safeTransfer(owner, tokenAmount);
    emit Recovered(tokenAddress, tokenAmount);
}
```

- **Incorrect require check**

```
354         // CHECK already boosted by same NFT contract??
355         require(boostedByNFT[msg.sender][_tokenId]==false,"Already boosted by this NFT");
356
357         require(totalNFTsBoostedBy[msg.sender]<=Erc1155Indexes.length,"Total boosts cannot be more than MAX NfT boosts available");
358
359         multiplierFactorNFT[msg.sender]= multiplierFactorNFT[msg.sender].add(multiplyFactor);
360
361         IERC1155(ERC1155AddressBooster).safeTransferFrom( msg.sender, devFundAdd, _tokenId, 1, "0x");
362
363         totalNFTsBoostedBy[msg.sender]=totalNFTsBoostedBy[msg.sender].add(1);
364         boostedByNFT[msg.sender][_tokenId] = true;
365
```

Function **boostByERC1155NFT** puts restrictions on maximum number of ERC1155 index boosters allowed per user by making sure that it should not exceed Erc1155Indexes array length i.e, total number of ERC1155 indexes available.

The check considers the existing value of **totalNFTsBoostedBy** instead of considering the new value, which will be **totalNFTsBoostedBy+1** after the successful operation, thus incorrect.

Although, it is not affecting the function's operational logic here, as it allows boosting of one ERC1155 index only once, so technically, the maximum possible ERC1155 index boosters per user, will be the number of ERC1155 indexes, i.e Erc1155Indexes array length itself.

Recommendation: Even though it is not affecting the operational logic here, consider adapting the appropriate checks.

Status: **Fixed**

PiStakingVault1

- Modifier **nonDuplicated** compares **poolExistence[_wantToken]** with a boolean constant, whereas **Boolean constants can be used directly and do not need to be compared to true or false.**

```
139       modifier nonDuplicated(address _wantToken) {
140           require(poolExistence[_wantToken] == false, "nonDuplicated: duplicated");
141           _;
142       }
```

Status: **Fixed**

- **Make sure the max_slots for a pool is set as a value more than max_per_user**

  Status: **Fixed**

- Consider reviewing operational logic for token swaps, and allow only non-zero token swaps

  Status: **Acknowledged**

- **Use of Strict Inequalities**

  [#L213]Function **deposit** uses strict inequalities for comparisons, as a consequence it may not be possible to completely utilize the upper bounds of compared values.

```
213     function deposit(uint256 _pid, uint256 totalNFTs, uint256[] memory _erc721tokenIds) public nonReentrant {
214         require(totalNFTs>0,"amount of totalNFTs to transfer cannot be 0");
215 
216         PoolInfo storage pool = poolInfo[_pid];
217         require(slots_filled[_pid]<pool.max_slots,"Max slots filled");
218 
219         uint256 _amount = totalNFTs.mul(pool.depositFeeNative);
220 
221         if(pool.isERC1155){
222             require(NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]<pool.max_per_user,"Max per user already done");
223 
224             IERC1155(pool.nativeNFTtoken).safeTransferFrom( msg.sender, address(this), pool.ERC1155NFTid, totalNFTs, "0x");
225             // donot duplicate erc1155 token ids
226             if(NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]==0){
227                 userInfo[_pid][msg.sender].AllNFTIds.push(pool.ERC1155NFTid);
228             }
229             NFTIdsDeposits[_pid][msg.sender][pool.ERC1155NFTid]+=totalNFTs;
230             slots_filled[_pid]+=totalNFTs;
231 
232         } else // erc721
233         {
234             require(userInfo[_pid][msg.sender].AllNFTIds.length<pool.max_per_user,"Max per user already done");
235             for(uint i=0;i<_erc721tokenIds.length;i++){
236                 IERC721(pool.nativeNFTtoken).transferFrom(msg.sender, address(this), _erc721tokenIds[i]);
237                 userInfo[_pid][msg.sender].AllNFTIds.push(_erc721tokenIds[i]);
```

Recommendation: Consider including equivalence with strict inequalities

Status: Fixed

- **Undocumented/Unused parameters of a Pool**

  Function add initializes a pool with multiple parameters. Some of which do not take part in any logical operation for the contract, and are highlighted below:

```
165         poolInfo.push(PoolInfo({
166             PiTokenAmtPerNFT : _piTokenAmtPerNFT,
167             nativeToken : _nativeToken,
168             nativeNFTtoken : _nativeNFTtoken,
169             ERC1155NFTid: _ERC1155NFTid,
170             isERC1155 : _isERC1155,
171             wantToken : _wantToken,
172             allocPoint : _allocPoint,
173             depositFeeNative : _depositFeeNative, // native tokens required as f
174             NFTperDepositFee : 1000, // eg. 1 NFT per 10 deposit fee
175             strat: _strat,
176             rewardPerLPTokenStored:0,
177             lastUpdateTime : block.timestamp,
178             max_slots : _max_slots,
179             max_per_user : _max_per_user
180         }));
```

  Recommendation: Consider reviewing and verifying the operational and business logic

  Status: Fixed

- Default visibility for state variable **timeLockInSecs** and mapping **lockTimeStamp** has been opted. (Commit: e19...c58)

```
109         uint256 timeLockInSecs;

110         mapping(address=>uint256) lockTimeStamp;
```

  Recommendation: Explicitly define visibility so as to avoid incorrect assumptions for access restrictions

  Status: Fixed

PiVaultStratLP and PiVaultStratToken

- **Missing Important checks in restakeAllToMigratedVault may cause imbalance and produce inappropriate results**

  function **restakeAllToMigratedVault** restakes want Tokens from existing PCSVaultAddress to **migrateVault.newPCSVaultAddress**. But the function lacks important checks, which may cause imbalance in the system.

  Some of the important checks are:
  - Make sure the **earnedAddress** i.e, the cake token stays the same for new or any PCSVault

  Recommendation: Consider adding the required checks.

  Status: Acknowledged

- **Commented Code:** The contracts contain some commented code.

```
300       // function buyBack(uint256 _earnedAmt) internal returns (uint256) {
301       //     if (buyBackRate <= 0) {
302       //         return _earnedAmt;
303       //     }
304       //
305       //     uint256 buyBackAmt = _earnedAmt.mul(buyBackRate).div(buyBackRateMax);
306       //
307       //     IERC20Upgradeable(earnedAddress).safeIncreaseAllowance(
308       //         uniRouterAddress,
309       //         buyBackAmt
310       //     );
311       //
312       //     IUniswapV2Router02(uniRouterAddress)
313       //         .swapExactTokensForTokensSupportingFeeOnTransferTokens(
314       //         buyBackAmt,
315       //         0,
316       //         earnedToPiPath,
317       //         buyBackAddress,
318       //         block.timestamp + 60
319       //     );
320       //
321       //     return _earnedAmt.sub(buyBackAmt);
322       // }
323
```

  Recommendation: Consider reviewing and verifying the operational and business logic

  Status: Acknowledged

Closing Summary

Several issues of high, medium and low severity have been reported during the audit.Majority of the issues has been fixed by the Pi Protocol team. Some suggestions are provided to improve the code quality.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the **Pi Protocol** Team. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the **Pi Protocol** Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
December, 2021

For

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com