BLOCKSEC

# Security Audit Report for Lista Dao Contracts

**Date:** September 3, 2025 **Version:** 1.0

**Contact:** contact@blocksec.com

# Contents

## Report Manifest

| Item | Description |
|---|---|
| Client | Lista |
| Target | Lista Dao Contracts |

## Version History

| Version | Date | Description |
|---|---|---|
| 1.0 | September 3, 2025 | First release |

## Signature

**About BlockSec** BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

# Chapter 1 Introduction

## 1.1 About Target Contracts

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

The target of this audit is the code repository [1] of Lista Dao Contracts of Lista.

Lista Dao is a protocol that enables users to earn rewards by strategically leveraging crypto assets. Users deposit collateral like BNB and ETH into the Interaction (CDP) module to borrow LisUSD. The borrowed LisUSD can be staked in the Jar Contract to accrue rewards, which are calculated by the ListaDistributor Contract. Additionally, users can stake BNB via the Stake Manager to receive slisBNB, earning bi-weekly LISTA rewards. slisBNB can then be used as collateral for further borrowing. Finally, locking LISTA tokens in the veLista contract grants eligibility for extra LISTA rewards. The protocol adds the support of PancakeSwap V3 LP as collateral, which is the main audit scope.

Note this audit only focuses on the smart contracts in the following directories/files:

- contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol
- contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingHub.sol
- contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingVault.sol
- contracts/ceros/provider/pancakeswapLpProvider/libraries/PcsV3LpLiquidationHelper.sol
- contracts/ceros/provider/pancakeswapLpProvider/libraries/PcsV3LpNumbersHelper.sol
- contracts/ceros/provider/LpUsd.sol
- contracts/ceros/provider/BaseTokenProvider.sol
- contracts/ceros/provider/PumpBTCProvider.sol
- contracts/ceros/provider/mBTCProvider.sol
- contracts/libraries/AuctionProxy.sol
- contracts/Interaction.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

[1] https://github.com/lista-dao/lista-dao-contracts

| Project | Version | Commit Hash |
|---|---|---|
| Lista Dao Contracts | Version 0 | 16d586fad26d457e1e58b1751db872515ac78bb7 |
| | Version 1 | d603a5c208cef3d0f49ce6896300701f7c5b27fe |
| | Version 2 | 75a12577fc2d64809dd98f524e473b3225e2a9f5 |

## 1.2 Disclaimer

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

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

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

## 1.3 Procedure of Auditing

We perform the audit according to the following procedure.

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

We show the main concrete checkpoints in the following.

### 1.3.1 Security Issues

- Access control
- Permission management
- Whitelist and blacklist mechanisms
- Initialization consistency
- Improper use of the proxy system
- Reentrancy
- Denial of Service (DoS)

- Untrusted external call and control flow
- Exception handling
- Data handling and flow
- Events operation
- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

### 1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

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

## 1.4 Security Model

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

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

**Table 1.1:** Vulnerability Severity Classification

|  |  | Likelihood |  |
|---|---|---|---|
|  |  | *High* | *Low* |
| **Impact** | *High* | High | Medium |
|  | *Low* | Medium | Low |

Accordingly, the severity measured in this report are classified into three categories: **High**,

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

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

**Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

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

# Chapter 2 Findings

In total, we found **thirteen** potential security issues. Besides, we have **three** recommendations and **four** notes.

- High Risk: 5
- Medium Risk: 1
- Low Risk: 7
- Recommendation: 3
- Note: 4

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | High | Incorrect sqrtPriceX96 calculation due to unhandled token decimal differences | Security Issue | Fixed |
| 2 | High | Fee drainage due to insufficient validation of providers | Security Issue | Fixed |
| 3 | High | Reward tokens are not sent to the recipient on liquidity decrease | Security Issue | Fixed |
| 4 | High | Lack of decimal conversion leads to incorrect liquidation calculations | Security Issue | Fixed |
| 5 | High | Incorrect calculation in the function _getMaxCdpWithdrawable() | Security Issue | Fixed |
| 6 | Medium | Rewards are incorrectly calculated when one of the token0 and token1 is the rewardToken | Security Issue | Fixed |
| 7 | Low | Interaction to MasterChefV3 still enabled in emergency mode | Security Issue | Fixed |
| 8 | Low | User liquidation record may never be deleted | Security Issue | Fixed |
| 9 | Low | Discrepancy between LP Token Value and LpUsd may cause auctions to fail to conclude | Security Issue | Fixed |
| 10 | Low | Potential circumvention of the pause mechanism | Security Issue | Fixed |
| 11 | Low | Lack of handling positions' rewards in emergency mode | Security Issue | Fixed |
| 12 | Low | Potential DoS during emergency operations | Security Issue | Fixed |
| 13 | Low | Providers did not burn ceToken and LpUsd in the function liquidation() | Security Issue | Fixed |
| 14 | - | Revise the annotations | Recommendation | Confirmed |
| 15 | - | Apply bound check for feeRate | Recommendation | Confirmed |

| | | | | |
|---|---|---|---|---|
| 16 | - | Lack of invoking function `_disableInitializers()` | Recommendation | Confirmed |
| 17 | - | LP value synchronization in `PancakeSwapV3LpProvider` should be in time | Note | - |
| 18 | - | The configuration of `providerCompatibilityMode` | Note | - |
| 19 | - | OpenZeppelin Initializable upgrade migration risks | Note | - |
| 20 | - | Potential centralization risks | Note | - |

The details are provided in the following sections.

## 2.1 Security Issue

### 2.1.1 Incorrect `sqrtPriceX96` calculation due to unhandled token decimal differences

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the contract `PcsV3LpNumbersHelper`, the `computeFairSqrtPriceX96()` function aims to calculate a fair market price based on oracle feeds. The implementation is flawed because it applies a static scaling factor to both token prices, which incorrectly assumes that both tokens have identical decimal precision. This approach fails to properly adjust the price ratio for tokens that possess different numbers of decimals. This miscalculation results in an inaccurate `sqrtPriceX96` value as the `sqrtPriceX96` should inherently incorporate the decimal differences between `token0` and `token1`.This can cause subsequent functions to compute incorrect token amounts and lead to a financial loss.

```
  uint160 sqrtPriceX96 = computeFairSqrtPriceX96(resilientOracle, token0, token1);
  uint160 sqrtPriceX96Lower = TickMath.getSqrtRatioAtTick(tickLower);
  uint160 sqrtPriceX96Upper = TickMath.getSqrtRatioAtTick(tickUpper);
  (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
    sqrtPriceX96, sqrtPriceX96Lower, sqrtPriceX96Upper, liquidity
  );
```

**Listing 2.1:** contracts/ceros/provider/pancakeswapLpProvider/libraries/PcsV3LpNumbersHelper.sol

```
function computeFairSqrtPriceX96(
  address resilientOracle,
  address token0,
  address token1
) private view returns (uint160 sqrtPriceX96) {
  // @note: ResilientOracle returns 8-decimal prices
  uint256 price0 = IResilientOracle(resilientOracle).peek(token0);
  uint256 price1 = IResilientOracle(resilientOracle).peek(token1);
  require(price0 != 0 && price1 != 0, "PcsV3LpNumbersHelper: zero-price");

  // scale both to 18 decimals (8 + 10)
  uint256 p0 = price0 * 1e10;
  uint256 p1 = price1 * 1e10;

  sqrtPriceX96 = toUint160(
    sqrt(_mul(p0, (1 << 96)) / p1) << 48
  );
}
```

**Listing 2.2:** contracts/ceros/provider/pancakeswapLpProvider/libraries/PcsV3LpNumbersHelper.sol

**Impact** The failure to properly handle token decimal differences leads to miscalculated liquidity amounts, creating a risk of financial loss for the protocol and its users.

**Suggestion** Revise the logic accordingly.

### 2.1.2 Fee drainage due to insufficient validation of `providers`

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the contract `PancakeSwapV3LpStakingVault`, the function `batchClaimRewards()` and its internal implementation `_batchClaimRewards()` facilitate reward distribution while deducting fees. However, due to insufficient validation of `provider` addresses in the input array, malicious actors can spoof `provider` addresses to return malformed amounts. As a result, malicious actors can steal the `availableFees` in the contract `PancakeSwapV3LpStakingVault`.

```
function batchClaimRewards(address[] memory providers, uint256[][] memory tokenIds) external whenNotPaused nonReentrant {
  _batchClaimRewards(msg.sender, providers, tokenIds);
}
```

**Listing 2.3:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingVault.sol

```
function _batchClaimRewards(address account, address[] memory providers, uint256[][] memory tokenIds) private {
  require(account != address(0), "PancakeSwapLpStakingVault: zero-address-provided");
  require(providers.length > 0, "PancakeSwapLpStakingVault: no-providers-provided");
  require(tokenIds.length == providers.length, "PancakeSwapLpStakingVault: tokenIds-length-mismatch");
  uint256 total;
  for (uint16 i = 0; i < providers.length; ++i) {
    uint256[] memory _tokenIds = tokenIds[i];
    require(_tokenIds.length > 0, "PancakeSwapLpStakingVault: no-tokenIds");
    uint256 amount = IPancakeSwapV3LpProvider(providers[i]).vaultClaimStakingReward(account, _tokenIds);
    // cut fee
    uint256 feeRate = feeRates[providers[i]];
    if (feeRate > 0) {
      uint256 fee = FullMath.mulDiv(amount, feeRate, DENOMINATOR);
      availableFees += fee;
      amount -= fee;
    }
    total += amount;
  }
  if (total > 0) {
    IERC20(rewardToken).safeTransfer(account, total);
  }
 }
```

**Listing 2.4:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingVault.sol

**Impact** Malicious actors can steal the `availableFees` in the `PancakeSwapV3LpStakingVault` contract.

**Suggestion** Apply validation for the parameter `providers`.

### 2.1.3 Reward tokens are not sent to the recipient on liquidity decrease

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The contract `PancakeSwapV3LpStakingHub` invokes the `decreaseLiquidity()` function in the contract `MasterChefV3` to remove liquidity from a position. However, in the contract `MasterChefV3` (code link), the function `decreaseLiquidity()` does not send the accrued rewards to the designated recipient address. Instead, the rewards are accumulated and stored in the user's position, which is a contradiction to the function's purpose. This issue prevents the protocol from correctly collecting the rewards. Moreover, the function `burn()` in the contract `MasterChefV3` requires the reward of the position to be 0 when burning the NFT token. Thus, this issue will also cause the transaction to revert when there exists rewards.

```
function _burnAndCollectTokens(
 uint256 tokenId,
 uint256 amount0Min,
 uint256 amount1Min
) internal returns (uint256 collectedAmount0WithFees, uint256 collectedAmount1WithFees) {
 address provider = msg.sender;
 IERC20 token0 = IERC20(IPancakeSwapV3LpProvider(provider).token0());
 IERC20 token1 = IERC20(IPancakeSwapV3LpProvider(provider).token1());

 uint256 preToken0Balance = token0.balanceOf(address(this));
 uint256 preToken1Balance = token1.balanceOf(address(this));

 // fully remove liquidity from the tokenId
 // after this tokens including fees are ready to collect
 IMasterChefV3(masterChefV3).decreaseLiquidity(
      IMasterChefV3.DecreaseLiquidityParams({
        tokenId: tokenId,
        liquidity: getLiquidity(tokenId),
        amount0Min: amount0Min,
        amount1Min: amount1Min,
        deadline: block.timestamp + 20 minutes // 20 minutes deadline
      })
    );
    // collect token0 and token1 including fees from the tokenId
    (uint256 collectedAmount0, uint256 collectedAmount1) = IMasterChefV3(masterChefV3).collect(
      IMasterChefV3.CollectParams({
        tokenId: tokenId,
        recipient: address(this),
        amount0Max: type(uint128).max, // just put max value to collect all fees
        amount1Max: type(uint128).max // just put max value to collect all fees
      })
    );
    // burn the LP
    IMasterChefV3(masterChefV3).burn(tokenId);
```

**Listing 2.5:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingHub.sol

**Impact** This issue prevents the protocol from correctly collecting the rewards. Moreover, the function burn() in the contract MasterChefV3 requires the reward of the position is 0 when burning the NFT token. Thus, this issue will also cause the transaction to revert when there exists rewards.

**Suggestion** Revise the logic accordingly.

### 2.1.4 Lack of decimal conversion leads to incorrect liquidation calculations

**Severity** High

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** The contract PancakeSwapV3LpProvider performs liquidation by first converting token0 and token1 to their USD values, which are then compared to the amount of LpUsd that needs to be paid. This calculation assumes that the token0Value and token1Value have the same decimal precision as the LpUsd amount, which is 1e18.

However, the values are not normalized to 1e18 before they are summed up, which leads to an incorrect comparison if the token0 and token1 have decimals different from 1e18. This issue can cause the protocol to over-liquidate a position, allowing a liquidator to acquire more collateral than they are entitled to.

```
    // after LP burn, recalculate token0 and token1 values
    token0Value = FullMath.mulDiv(record.token0Left, token0Price, RESILIENT_ORACLE_DECIMALS);
    token1Value = FullMath.mulDiv(record.token1Left, token1Price, RESILIENT_ORACLE_DECIMALS);
    // make sure enough tokens to cover the amount
    require((token0Value + token1Value) >= amount, "PcsV3LpProvider: insufficient-lp-value");
    // step 5. pay by tokens
    PcsV3LpLiquidationHelper.PaymentParams memory paymentParams = PcsV3LpLiquidationHelper.PaymentParams({
      recipient: recipient,
      amountToPay: amount,
      token0: token0,
      token1: token1,
      token0Value: token0Value,
      token1Value: token1Value,
      token0Left: record.token0Left,
      token1Left: record.token1Left
    });
    // leftover record will be updated after
    (uint256 newToken0Left, uint256 newToken1Left) = PcsV3LpLiquidationHelper.payByToken0AndToken1(paymentParams);
    // update leftover record
```

**Listing 2.6:**
contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

```
// Remaining amount (in USD) that needs to be paid
uint256 amountLeft = amountToPay;

// pay with token0 first
if (amount0 > 0 && token0Value > 0) {
  uint256 token0MaxPayable = token0Value;

  if (token0MaxPayable >= amountLeft) {
    // token0 alone is enough to pay the required amount
    uint256 token0AmountToSend = FullMath.mulDiv(amountLeft, amount0, token0MaxPayable);
    IERC20(token0).safeTransfer(recipient, token0AmountToSend);
    token0Sent = token0AmountToSend;
    amountLeft = 0;
  } else {
    // send all of token0
    IERC20(token0).safeTransfer(recipient, amount0);
    token0Sent = amount0;
    amountLeft -= token0MaxPayable;
  }
}

// pay the remainder with token1 if necessary
if (amountLeft > 0 && amount1 > 0 && token1Value > 0) {
  uint256 token1MaxPayable = token1Value;
  uint256 token1AmountToSend = FullMath.mulDiv(amountLeft, amount1, token1MaxPayable);
  IERC20(token1).safeTransfer(recipient, token1AmountToSend);
  token1Sent = token1AmountToSend;
  amountLeft = 0;
}
```

**Listing 2.7:**
contracts/ceros/provider/pancakeswapLpProvider/libraries/PcsV3LpLiquidationHelper.sol

**Impact** This issue can cause the protocol to over-liquidate a position, allowing a liquidator to acquire more collateral than they are entitled to.

**Suggestion** Revise the logic accordingly.

### 2.1.5 Incorrect calculation in the function `_getMaxCdpWithdrawable()`

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the contract `PancakeSwapV3LpProvider`, the `_getMaxCdpWithdrawable()` function is used to calculate the maximum amount a user can withdraw from their collateral `LpUsd`.

Specifically, at line 648, the variable `collateralValue` represents the value of `lpUsd` locked by the `user`. However, at line 654, `art` does not represent the total debt of the `user` but rather the total debt of all users under that collateral type.

Two scenarios may arise:

Case 1: Due to the total debt being excessively large, the function incorrectly returns zero due to `if (collateralValue <= minRequiredCollateral) return 0;`.

Case 2: Although the total debt is relatively small, which still exceeds the `user`'s debt, causing the returned value (i.e., `return collateralValue - minRequiredCollateral;)` to be smaller than it should be.

This issue affects the behavior of the following two functions:

1. In the function `_syncUserCdpPosition()`, since `withdrawableLpUsd` is calculated to be smaller than its correct value, `burnAmount` will also be understated, which will cause less `LpUsd` being burned.

2. In the function `release()`, it uses `_getMaxCdpWithdrawable()` to determine how much collateral the user can release. This issue may cause the user unable to withdraw their collateral.

```
646 function _getMaxCdpWithdrawable(address user) internal returns (uint256 withdrawableAmount) {
647   // get collateralized LpUSD
648   uint256 collateralValue = ICdp(cdp).locked(lpUsd, user);
649   // get ilk
650   (,bytes32 ilk,,) = ICdp(cdp).collaterals(lpUsd);
651   // refresh interest
652   ICdp(cdp).drip(address(lpUsd));
653   // get rate
654   (uint256 art, uint256 rate,,,) = ICdp(cdp).vat().ilks(ilk);
655   // get debt
656   uint256 debt = FullMath.mulDiv(art, rate, RAY);
657   // get MCR
658   (,uint256 mat) = ICdp(cdp).spotter().ilks(ilk);
659   // calculate the minimum required collateral
660   uint256 minRequiredCollateral = FullMath.mulDiv(debt, mat, RAY);
661   // if collateral value is less than or equal to the minimum required collateral,
662   if (collateralValue <= minRequiredCollateral) return 0;
663   // calculate the withdrawable amount
664   withdrawableAmount = collateralValue - minRequiredCollateral;
665 }
```

**Listing 2.8:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

```solidity
function _syncUserCdpPosition(address user, bool syncLpPrice) internal {
  // sync current user Lp total value
  uint256 _userLpTotalValue = _syncUserLpTotalValue(user, syncLpPrice);
  // convert with lpDiscountRate
  _userLpTotalValue = FullMath.mulDiv(_userLpTotalValue, lpDiscountRate, DENOMINATOR);
  // get total deposited LP_USD amount in the cdp
  uint256 totalLpUsd = ICdp(cdp).locked(lpUsd, user);
  // if user has more LP value than the total LP_USD amount in the cdp
  if (_userLpTotalValue > totalLpUsd) {
    // mint LP_USD
    uint256 mintAmount = _userLpTotalValue - totalLpUsd;
    ILpUsd(lpUsd).mint(address(this), mintAmount);
    ILpUsd(lpUsd).approve(cdp, mintAmount);
    // deposit the difference to cdp
    ICdp(cdp).deposit(user, lpUsd, mintAmount);
  } else if (_userLpTotalValue < totalLpUsd) {
    // if user has less LP value than the total LP_USD amount in the cdp,
    // burn LP_USD from the user
    uint256 burnAmount = totalLpUsd - _userLpTotalValue;
    uint256 withdrawableLpUsd = _getMaxCdpWithdrawable(user);
    // if burn amount is more than the withdrawable amount
    // we withdraw as much as we can, the position should be liquidated very soon
    if (burnAmount > withdrawableLpUsd) {
      burnAmount = withdrawableLpUsd;
      // notify our liquidator to kickoff the liquidation
      emit Liquidatable(
        user,
        userTotalLpValue[user],
        totalLpUsd
      );
    }
    // update cdp position
    ICdp(cdp).withdraw(user, lpUsd, burnAmount);
    ILpUsd(lpUsd).burn(address(this), burnAmount);
  }
  emit UserCdpPositionSynced(
    user,
    _userLpTotalValue,
    ICdp(cdp).locked(lpUsd, user)
  );
}
```

**Listing 2.9:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

```solidity
function release(uint256 tokenId) override external nonReentrant whenNotPaused {
  // check if the caller is the owner of the LP token
  address user = msg.sender;
  require(lpOwners[tokenId] == user, "PcsV3LpProvider: not-token-owner");
  require(!userLiquidations[user].ongoing, "PcsV3LpProvider: liquidation-ongoing");

  // fully sync. user CDP position
  _syncUserCdpPosition(user, true);
  uint256 withdrawableAmount = _getMaxCdpWithdrawable(user);
  uint256 wishToWithdraw = FullMath.mulDiv(lpValues[tokenId], lpDiscountRate, DENOMINATOR);
  require(wishToWithdraw <= withdrawableAmount, "PcsV3LpProvider: lp-value-exceeds-withdrawable-amount");

  // withdraw from Staking Hub with the harvested rewards
  uint256 rewardAmount = IPancakeSwapV3LpStakingHub(pancakeStakingHub).withdraw(tokenId);
  // remove token
  _removeToken(user, tokenId);
  // sync user position
  _syncUserCdpPosition(msg.sender, false);
  // send reward and cut fee
  _sendRewardAfterFeeCut(rewardAmount, user);
  // transfer LP token back to the user
  IERC721(nonFungiblePositionManager).safeTransferFrom(address(this), user, tokenId);

  emit WithdrawLp(msg.sender, tokenId, lpValues[tokenId]);
}
```

**Listing 2.10:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

**Impact** This issue affects the behavior of the following two functions: 1. In the function `_syncUserCdpPosition()`, since `withdrawableLpUsd` is calculated to be smaller than its correct value, `burnAmount` will also be understated, which will cause less `LpUsd` being burned. 2. In the function `release()`, it uses `_getMaxCdpWithdrawable()` to determine how much collateral the user can release. This issue may cause the user unable to withdraw their collateral.

**Suggestion** Revise the logic accordingly.

### 2.1.6 Rewards are incorrectly calculated when one of the `token0` and `token1` is the `rewardToken`

**Severity** Medium

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The function `burnAndCollect()` is designed to handle the collection of `token0`, `token1` and rewards. The function calculates the reward amount by taking the difference in the contract's balance of the `rewardToken` before and after the internal `_burnAndCollectTokens()` function is called. If the `rewardToken` happens to be the same as either `token0` or `token1`, its value will be included in the `amount0` or `amount1` returned from the internal call, which leads to an inaccurate reward calculation. This issue results in a portion of the rewards being incorrectly transferred as `token0` or `token1` instead of as rewards.

This issue has a high probability of occurring, as many PancakeSwap V3 pools have one of `token0` and `token1` as `CAKE`, which is also the reward token.

```
function burnAndCollect(uint256 tokenId, uint256 amount0Min, uint256 amount1Min)
override
external
checkTokenIdWithProvider(tokenId)
onlyProvider
whenNotPaused
nonReentrant
returns (uint256 amount0, uint256 amount1, uint256 rewards) {
  require(tokenId > 0, "PancakeSwapStakingHub: non-zero-tokenId");
  // pre-balance of reward
  uint256 preRewardBalance = IERC20(rewardToken).balanceOf(address(this));
  // decrease liquidity then burn LP
  // @note will harvest reward as well
  (amount0, amount1) = _burnAndCollectTokens(
    tokenId,
    amount0Min,
    amount1Min
  );
  // get rewards amount
  rewards = IERC20(rewardToken).balanceOf(address(this)) - preRewardBalance;
  // send rewards to provider
  if (rewards > 0) {
    IERC20(rewardToken).safeTransfer(msg.sender, rewards);
    emit Harvest(msg.sender, tokenId, rewards);
  }
  // remove tokenId record
  _removeTokenRecord(tokenId);
  // emit event
  emit BurnLp(msg.sender, tokenId, rewards, amount1, amount0);
}
```

**Listing 2.11:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingHub.sol

**Impact** This issue results in a portion of the rewards being incorrectly transferred as `token0` or `token1` instead of as rewards.

**Suggestion** Revise the logic accordingly.

### 2.1.7 Interaction to `MasterChefV3` still enabled in emergency mode

**Severity** Low

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the contract `PancakeSwapV3LpStakingHub`, the function `emergencyWithdraw()` enables the `MANAGER` to activate `emergencyMode` and withdraw all staked position NFTs from `MasterChefV3`. However, several functions including `deposit()` and `burnAndCollect()` maintain active interaction with `MasterChefV3` even when `emergencyMode` is on.

```
function deposit(uint256 tokenId) override external onlyProvider whenNotPaused nonReentrant {
  require(tokenId > 0, "PancakeSwapStakingHub: non-zero-tokenId");
```

```
    address provider = msg.sender;
    // transfer token from provider to MasterChefV3
    IERC721(nonFungiblePositionManager).safeTransferFrom(provider, address(this), tokenId);
    // transfer to MasterChefV3
    IERC721(nonFungiblePositionManager).safeTransferFrom(address(this), masterChefV3, tokenId);
    // record tokenId
    tokenIds.push(tokenId);
    tokenIdToProvider[tokenId] = provider;

    emit DepositLp(provider, tokenId);
  }
```

**Listing 2.12:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingHub.sol

**Impact** Even when emergencyMode is activated, several functions including deposit() and burnAndCollect() continue to interact with MasterChefV3.

**Suggestion** Revise the logic of the functions deposit() and burnAndCollect().

### 2.1.8 User liquidation record may never be deleted

**Severity** Low

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** The function buyFromAuction() invokes the helioProvider.liquidation() function by setting the isLeftOver to be true, to delete a user's liquidation record, when the leftover amount is greater than 0. However, according to the function take() in the contract clip, an auction will also be concluded, when the lot is 0 with the leftover amount is 0. In this scenario, the helioProvider.liquidation() function will not be invoked with isLeftOver set to true, which prevents the user's liquidation record from ever being deleted while the auction has been concluded.

```
    leftover = vat.gem(collateral.ilk, urn); // userGemBalanceBefore
    ClipperLike(collateral.clip).take(param.auctionId, param.collateralAmount, param.maxPrice,
        address(this), "");
    leftover = vat.gem(collateral.ilk, urn) - leftover; // leftover

    collateral.gem.exit(address(this), vat.gem(collateral.ilk, address(this)));
    hayJoin.exit(address(this), vat.hay(address(this)) / RAY);

    // Balances rest
    hayBal = hay.balanceOf(address(this)) - hayBal;
    gemBal = collateral.gem.gem().balanceOf(address(this)) - gemBal;
    hay.safeTransfer(param.receiverAddress, hayBal);

    vat.nope(address(collateral.clip));

    if (address(helioProvider) != address(0)) {
      IERC20Upgradeable(collateral.gem.gem()).safeTransfer(address(helioProvider), gemBal);
```

```
  helioProvider.liquidation(urn, param.receiverAddress, gemBal, data, false); // Burn router ceToken and mint abnbc to receiver

  if (leftover != 0) {
   // Auction ended with leftover
   vat.flux(collateral.ilk, urn, address(this), leftover);
   collateral.gem.exit(address(helioProvider), leftover); // Router (disc) gets the remaining ceabnbc
   helioProvider.liquidation(urn, param.receiverAddress, leftover, data, true); // Router burns them and gives abnbc remaining
  }
 } else {
```

**Listing 2.13:** contracts/libraries/AuctionProxy.sol

```

   if (lot == 0) {
     _remove(id);
   } else if (tab == 0) {
     vat.flux(ilk, address(this), usr, lot);
     _remove(id);
   } else {
```

**Listing 2.14:** contracts/clip.sol

```
function liquidation(
 address owner,
 address recipient,
 uint256 amount,
 bytes memory data,
 bool isLeftOver
) external nonReentrant onlyCdp {
 require(owner != address(0), "PcsV3LpProvider: invalid-owner");
 require(recipient != address(0), "PcsV3LpProvider: invalid-recipient");
 require(amount > 0, "PcsV3LpProvider: invalid-amount");
 // get user token0 and token1 leftover from previous liquidation(if any)
 UserLiquidation storage record = userLiquidations[owner];
 // liquidation ended, send leftover tokens and LP to the owner
 if (isLeftOver) {
  // sweep the leftover lpUsd at cdp after liquidation
  PcsV3LpLiquidationHelper.sweepLeftoverLpUsd(
   owner,
   lpUsd,
   cdp
  );
  if (userLps[owner].length > 0) {
   // re-init user’s position at CDP
   _syncUserCdpPosition(owner, true);
  }
  // returns all leftover token0 and token1 to user
  if (record.token1Left > 0) {
   IERC20(token1).safeTransfer(owner, record.token1Left);
  }
  if (record.token0Left > 0) {
    IERC20(token0).safeTransfer(owner, record.token0Left);
  }
  // delete user’s liquidation record
  delete userLiquidations[owner];
```

Listing 2.15:
contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

**Impact** This prevents the user’s liquidation record from ever being deleted while the auction has been concluded.

**Suggestion** Revise the logic accordingly.

### 2.1.9 Discrepancy between LP Token Value and `LpUsd` may cause auctions to fail to conclude

**Severity** Low

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** When the function `startAuction()` in the contract `Interaction` is called, it records the user’s collateral amount as `lot` in the auction and begins the auction process. Under normal circumstances, like `mBTCProvider`, the collateral `ceToken` can be exchanged for `mBTC` on a 1:1 basis, even if the `mBTC` price falls.

However, this mechanism fails when PancakeV3 LP tokens are used as collateral. After the function `startAuction()` is invoked, if the value of the PancakeV3 LP tokens continues to fall, the `LpUsd` recorded as the `lot` will result in higher value than the actual value of the LP tokens. In this scenario, even though the auction `lot` is not zero, the `LpUsd` obtained after exiting the `lot` are unable to redeem `token0` and `token1` from the contract `PancakeSwapV3LpProvider` due to insufficiency. Resetting the auction with the function `resetAuction()` to update the `LpUsd` price does not solve this issue.

```
function bark(bytes32 ilk, address urn, address kpr) external auth returns (uint256 id) {
    require(live == 1, "Dog/not-live");

    (uint256 ink, uint256 art) = vat.urns(ilk, urn);
    Ilk memory milk = ilks[ilk];
    uint256 dart;
    uint256 rate;
    uint256 dust;
    {
        uint256 spot;
        (,rate, spot,, dust) = vat.ilks(ilk);
        require(spot > 0 && mul(ink, spot) < mul(art, rate), "Dog/not-unsafe");

        // Get the minimum value between:
        // 1) Remaining space in the general Hole
        // 2) Remaining space in the collateral hole
        require(Hole > Dirt && milk.hole > milk.dirt, "Dog/liquidation-limit-hit");
        uint256 room = min(Hole - Dirt, milk.hole - milk.dirt);
```

```

            // uint256.max()/(RAD*WAD) = 115,792,089,237,316
            dart = min(art, mul(room, WAD) / rate / milk.chop);

            // Partial liquidation edge case logic
            if (art > dart) {
                if (mul(art - dart, rate) < dust) {

                    // If the leftover Vault would be dusty, just liquidate it entirely.
                    // This will result in at least one of dirt_i > hole_i or Dirt > Hole becoming true.
                    // The amount of excess will be bounded above by ceiling(dust_i * chop_i / WAD).
                    // This deviation is assumed to be small compared to both hole_i and Hole, so that
                    // the extra amount of target HAY over the limits intended is not of economic concern.
                    dart = art;
                } else {

                    // In a partial liquidation, the resulting auction should also be non-dusty.
                    require(mul(dart, rate) >= dust, "Dog/dusty-auction-from-partial-liquidation");
                }
            }
        }

        uint256 dink = mul(ink, dart) / art;

        require(dink > 0, "Dog/null-auction");
        require(dart <= 2**255 && dink <= 2**255, "Dog/overflow");

        vat.grab(
            ilk, urn, milk.clip, address(vow), -int256(dink), -int256(dart)
        );

        uint256 due = mul(dart, rate);

        {   // Avoid stack too deep
            // This calcuation will overflow if dart*rate exceeds ~10^14
            uint256 tab = mul(due, milk.chop) / WAD;
            Dirt = add(Dirt, tab);
            ilks[ilk].dirt = add(milk.dirt, tab);

            id = ClipperLike(milk.clip).kick({
                tab: tab,
                lot: dink,
                usr: urn,
                kpr: kpr
            });
        }

        emit Bark(ilk, urn, dink, dart, due, milk.clip, id);
    }
```

Listing 2.16: contracts/dog.sol

**Impact** The auction may never be closed.

**Suggestion** Revise the logic accordingly.

### 2.1.10 Potential circumvention of the pause mechanism

**Severity** Low

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** In the contract PancakeSwapV3LpProvider, the function onERC721Received() processes PancakeSwap V3 Positions deposits without checking the pause status, potentially compromising its intended security mechanism. Specifically, users can directly transfer Positions to the contract while it is paused. As a result, this can cause a circumvention of the pause mechanism.

```
  address /*operator*/,
  address from,
  uint256 tokenId,
  bytes calldata /*data*/
) external returns (bytes4) {
  // only accept NFT sent from NonFungiblePositionManager
```

Listing 2.17: contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

**Impact** The pause mechanism in the contract PancakeSwapV3LpProvider can be circumvented.

**Suggestion** Apply whenNotPaused modifier to the function onERC721Received().

### 2.1.11 Lack of handling positions' rewards in emergency mode

**Severity** Low

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** The function emergencyWithdraw() enables the MANAGER to set the emergencyMode to true for the contract and withdraw all staked positions from MasterChefV3 for users. Although MasterChefV3 ceases accruing new rewards for positions during its emergency state, rewards for a specific tokenId may not be zero due to rewards accumulated historically.

However, the received rewards are neither transferred to users nor properly recorded. This could lead to unfair rewards distribution.

```
function emergencyWithdraw() external nonReentrant onlyRole(MANAGER) {
  require(!emergencyMode, "PancakeSwapStakingHub: already-in-emergency-mode");
  require(IMasterChefV3(masterChefV3).emergency(), "PancakeSwapStakingHub: masterChefV3-not-in-emergency-mode");
  emergencyMode = true;
  // withdraw all tokenIds from MasterChefV3
  for (uint256 i = 0; i < tokenIds.length; i++) {
    IMasterChefV3(masterChefV3).withdraw(tokenIds[i], address(this));
  }
  emit EmergencyWithdraw();
}
```

Listing 2.18: contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingHub.sol

```
function stopEmergencyMode() external nonReentrant onlyRole(MANAGER) {
  require(emergencyMode, "PancakeSwapStakingHub: not-in-emergency-mode");
  require(!IMasterChefV3(masterChefV3).emergency(), "PancakeSwapStakingHub: masterChefV3-is-in-
      emergency-mode");
  emergencyMode = false;
  // transfer all LP back to MasterChefV3 for farming
  for (uint256 i = 0; i < tokenIds.length; i++) {
    // transfer token to MasterChefV3
    IERC721(nonFungiblePositionManager).safeTransferFrom(address(this), masterChefV3, tokenIds[i
        ]);
  }
  emit StopEmergencyMode();
}
```

Listing 2.19: contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingHub.sol

**Impact** Unfair rewards distribution due to lack of handling positions' rewards.

**Suggestion** Revise the logic accordingly.

### 2.1.12 Potential DoS during emergency operations

**Severity** Low

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** In the contract PancakeSwapV3LpStakingHub, both the stopEmergencyMode() and emergencyWithdraw() functions iterate through the tokenIds array to transfer tokens. As the array grows with more deposits, these operations may exceed the gas limit. As a result, this could result in potential DoS during emergency operations.

Additionally, if the contract MasterChefV3 is in an emergency state due to an attack and some of NFT tokens are stolen, the loop in the function emergencyWithdraw() may fail entirely, preventing the withdrawal of non-stolen tokens.

```
function stopEmergencyMode() external nonReentrant onlyRole(MANAGER) {
  require(emergencyMode, "PancakeSwapStakingHub: not-in-emergency-mode");
  require(!IMasterChefV3(masterChefV3).emergency(), "PancakeSwapStakingHub: masterChefV3-is-in-
      emergency-mode");
  emergencyMode = false;
  // transfer all LP back to MasterChefV3 for farming
  for (uint256 i = 0; i < tokenIds.length; i++) {
    // transfer token to MasterChefV3
    IERC721(nonFungiblePositionManager).safeTransferFrom(address(this), masterChefV3, tokenIds[i]);
  }
  emit StopEmergencyMode();
}
```

**Listing 2.20:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingHub.sol

```
function emergencyWithdraw() external nonReentrant onlyRole(MANAGER) {
  require(!emergencyMode, "PancakeSwapStakingHub: already-in-emergency-mode");
  require(IMasterChefV3(masterChefV3).emergency(), "PancakeSwapStakingHub: masterChefV3-not-in-emergency-mode");
  emergencyMode = true;
  // withdraw all tokenIds from MasterChefV3
  for (uint256 i = 0; i < tokenIds.length; i++) {
    IMasterChefV3(masterChefV3).withdraw(tokenIds[i], address(this));
  }
  emit EmergencyWithdraw();
}
```

**Listing 2.21:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingHub.sol

**Impact** Potential DoS during emergency operations.

**Suggestion** Revise the logic accordingly.

### 2.1.13 Providers did not burn `ceToken` and `LpUsd` in the function `liquidation()`

**Severity** Low

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The function `buyFromAuction()` is designed to handle the liquidation process by transferring the gem tokens to the respective provider, which is then responsible for distributing the underlying collateral. This process works correctly for the `BaseTokenProvider`, where the gem tokens are the actual collateral token.

For other providers, however, this process is flawed.

1.The `mBTCProvider` and `PumpBTCProvider` receive `ceToken`, but they transfer the real collateral token without burning the `ceToken`.

2.The `PancakeSwapV3LpProvider` receives `LpUsd` tokens, but it transfers the LP underlying tokens(i.e., `token0` and `token1`) to the receiver without burning the `LpUsd` tokens.

This critical flaw causes the supply of `ceToken` and `LpUsd` to exceed the value of their corresponding collateral.

```
  if (address(helioProvider) != address(0)) {
    IERC20Upgradeable(collateral.gem.gem()).safeTransfer(address(helioProvider), gemBal);
    helioProvider.liquidation(urn, param.receiverAddress, gemBal, data, false); // Burn router ceToken and mint abnbc to receiver
    if (leftover != 0) {
      // Auction ended with leftover
      vat.flux(collateral.ilk, urn, address(this), leftover);
      collateral.gem.exit(address(helioProvider), leftover); // Router (disc) gets the remaining ceabnbc
      helioProvider.liquidation(urn, param.receiverAddress, leftover, data, true); // Router burns them and gives abnbc remaining
    }
```

**Listing 2.22:** contracts/libraries/AuctionProxy.sol

```
function liquidation(
  address _recipient,
  uint256 _lpAmount
) public virtual nonReentrant whenNotPaused onlyRole(PROXY) {
  require(_recipient != address(0));
  uint256 _amount = _lpAmount / scale;
  IERC20(token).safeTransfer(_recipient, _amount);

  emit Liquidation(_recipient, _amount, _lpAmount);
}
```

**Listing 2.23:** contracts/ceros/provider/mBTCProvider.sol

```
function liquidation(
  address _recipient,
  uint256 _lpAmount
) public virtual nonReentrant whenNotPaused onlyRole(PROXY) {
  require(_recipient != address(0));
  uint256 _amount = _lpAmount / scale;
  IERC20(token).safeTransfer(_recipient, _amount);

  emit Liquidation(_recipient, _amount, _lpAmount);
}
```

**Listing 2.24:** contracts/ceros/provider/PumpBTCProvider.sol

```
function liquidation(
  address owner,
  address recipient,
  uint256 amount,
  bytes memory data,
  bool isLeftOver
) external nonReentrant onlyCdp {
  require(owner != address(0), "PcsV3LpProvider: invalid-owner");
  require(recipient != address(0), "PcsV3LpProvider: invalid-recipient");
  require(amount > 0, "PcsV3LpProvider: invalid-amount");
```

**Listing 2.25:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

**Impact** This critical flaw causes the supply of `ceToken` and `LpUsd` to exceed the value of their corresponding collateral.

**Suggestion** Revise the logic accordingly.

## 2.2 Recommendation

### 2.2.1 Revise the annotations

**Status** Confirmed

**Introduced by** `Version 1`

**Description** The following annotations are either incorrect or inconsistent. Thus, it is recommended to revise them for better code readability and clarity.

1.In the contract `PumpBTCProvider`, the token used is `PumpBTC`, but the annotation mistakenly refers to it as `bumpBTC`.

```
// 1. transfer bumpBTC to provider
```

**Listing 2.26:** contracts/ceros/provider/PumpBTCProvider.sol

**Suggestion** Revise the annotations accordingly.

**Feedback from the project** The project will revise the annotations in the future.

### 2.2.2 Apply bound check for `feeRate`

**Status** Confirmed

**Introduced by** `Version 1`

**Description** In the function `registerLpProvider()`, the `feeRate` lacks validation to ensure they are not larger than `DENOMINATOR`. It is recommended to add such validation to prevent potential misoperation.

```
function registerLpProvider(address provider, uint256 feeRate) external onlyRole(MANAGER) {
  require(
    provider != address(0) &&
    !lpProviders[provider],
    "PancakeSwapLpStakingVault: provider-already-registered"
  );
  lpProviders[provider] = true;
  feeRates[provider] = feeRate;
  emit LpProviderRegistered(provider, feeRate);
}
```

**Listing 2.27:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpStakingVault.sol

**Suggestion** Apply bound check for the parameter `feeRate`.

**Feedback from the project** The `feeRate` is set by a 3/6 threshold multi-sig wallet, so it's under control securely.

### 2.2.3 Lack of invoking function `_disableInitializers()`

**Status** Confirmed

**Introduced by** `Version 1`

**Description** The `_disableInitializers()` function is not invoked in the constructor of the contracts `mBTCProvider`, `PumpBTCProvider`, and `Interaction`. Invoking this function prevents the contract itself from being initialized, thereby avoiding unexpected behaviors.

**Suggestion** Invoke the function `_disableInitializers()` in the constructor.

**Feedback from the project** This project will address this in the future.

## 2.3 Note

### 2.3.1 LP value synchronization in `PancakeSwapV3LpProvider` should be in time

**Introduced by** `Version 1`

**Description** In the contract `PancakeSwapV3LpProvider`, the function `_deposit()` only synchronizes the LP valuation for the newly deposited NFT without synchronizing previously deposited NFTs. Moreover, the function `poke()` in the contract `Interaction` contract assumes collateral assets have a fixed quantity and only updates the collateral assets' prices. This assumption is not suitable for PancakeSwap V3-based collateral, where the collateral value is determined by the amount of `LpUsd`, whose price is always one dollar.

Therefore, LP valuations must be properly monitored and synchronized via off-chain programs by invoking the functions `syncUserLpValues()` and `batchSyncUserLpValues()` to prevent outdated LP valuations.

```
function _deposit(address user, uint256 tokenId) internal {
  // check if user has reached the max LP limit
  require(userLps[user].length <= maxLpPerUser, "PcsV3LpProvider: max-lp-reached");

  // get lp value and verify the underlying price
  uint256 lpValue = _syncLpValue(tokenId);
  require(lpValue >= minLpValue, "PcsV3LpProvider: min-lp-value-not-met");

  // update lpOwners, lpValues
  lpOwners[tokenId] = user;
  userLps[user].push(tokenId);
  // farm LP by deposit to pancakeStakingHub
  IERC721(nonFungiblePositionManager).approve(pancakeStakingHub, tokenId);
  IPancakeSwapV3LpStakingHub(pancakeStakingHub).deposit(tokenId);
  // update user position
  _syncUserCdpPosition(user, false);

  emit DepositLp(user, tokenId, lpValue);
}
```

**Listing 2.28:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

```
function _syncUserLpTotalValue(address user, bool syncLpPrice) internal returns (uint256 userLpTotalValue) {
  // reset userLpTotalValue
  userLpTotalValue = 0;
  // iterate through user’s LPs and sum up the appraised value
  uint256[] storage userLpTokens = userLps[user];
  for (uint256 i = 0; i < userLpTokens.length; i++) {
    uint256 tokenId = userLpTokens[i];
    uint256 lpValue = syncLpPrice ? _syncLpValue(tokenId) : lpValues[tokenId];
    userLpTotalValue += lpValue;
  }
  // update user’s total LP value
  userTotalLpValue[user] = userLpTotalValue;
}
```

**Listing 2.29:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

```
  function poke(address token) public {
      CollateralType memory collateralType = collaterals[token];
      _checkIsLive(collateralType.live);

      spotter.poke(collateralType.ilk);
  }
```

**Listing 2.30:** contracts/Interaction.sol

```
  function poke(bytes32 ilk) external {
      (bytes32 val, bool has) = ilks[ilk].pip.peek();
      uint256 spot = has ? rdiv(rdiv(mul(uint(val), 10 ** 9), par), ilks[ilk].mat) : 0;
      vat.file(ilk, "spot", spot);
      emit Poke(ilk, val, spot);
  }
```

**Listing 2.31:** contracts/spot.sol

```
function peek() override public pure returns (bytes32, bool) {
  // returns in 18 decimals
  return (bytes32(uint(1e18)), true);
}
```

**Listing 2.32:** contracts/ceros/provider/pancakeswapLpProvider/PancakeSwapV3LpProvider.sol

### 2.3.2 The configuration of `providerCompatibilityMode`

**Introduced by** `Version 1`

**Description** For the `PancakeSwapV3LpProvider`, it is critical to keep its mapping `providerCompatibilityMode` set to `false`. This configuration ensures that users cannot withdraw `LpUsd` without interacting through the provider.

```
function withdraw(
    address participant,
    address token,
    uint256 dink
) external nonReentrant returns (uint256) {
    CollateralType memory collateralType = collaterals[token];
    _checkIsLive(collateralType.live);

    drip(token);
    poke(token);
    if (helioProviders[token] != address(0)) {
        if (providerCompatibilityMode[token]) {
            require(
                msg.sender == participant || msg.sender == helioProviders[token],
                "Interaction/Caller must be participant/provider"
            );
```

**Listing 2.33:** contracts/Interaction.sol

### 2.3.3 OpenZeppelin Initializable upgrade migration risks

**Introduced by** `Version 1`

**Description** The project currently uses OpenZeppelin's Initializable contract (v4.8.3) to implement upgradeable contracts. It is important to note that subsequent versions (v5.0.0+) implement ERC-7201 namespaced storage to address potential storage collision risks.. This change relocates initialization state variables from direct storage slots (e.g.,`_initialized`) to namespaced storage structures (e.g., `$._initialized`). When upgrading to newer Initializable versions, the project must ensure proper migration of initialization states to prevent contracts from being reinitialized, which could lead to severe security vulnerabilities including state corruption and unauthorized access.

### 2.3.4 Potential centralization risks

**Introduced by** `Version 1`

**Description** In this project, several privileged roles (e.g., `MANAGER`, `wards[usr]`) can conduct sensitive operations, which introduces potential centralization risks. For example, `wards[usr]` can set collateral information in the contract `Interaction` based on the protocol. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS