2pi Network Loans

Security Report

coinspect

2PI Network Loans

Smart Contract Audit

V230410 Prepared for 2PI Network • April 2023

1. Executive Summary

In **January 2023, 2PI Network** engaged Coinspect to perform a source code review of **2PI Network Loans**. The objective of the project was to evaluate the security of the smart contracts.

Several high risk vulnerabilities and issues were identified during this audit that put users or the protocol at risk. Coinspect recommends the protocol is re-audited once the issues are resolved before deploying or going to production phase.

The following issues were identified during the initial assessment:

High Risk	Medium Risk	Low Risk
Open **0**	Open **0**	Open **0**
Fixed 2	Fixed 5	Fixed 3
Reported 4	Reported 6	Reported 4

Coinspect identified four high-risk issues, seven medium-risk issues and three low-risk issues.

The most important issues reported are related to:

- Users getting less assets than expected while withdrawing from a collateral pool (**PIN-10**).
- Unprofitable liquidations under congested network conditions (**PIN-11**).
- Collateral pool's controller assuming that strategies will always be profitable (**PIN-12**).
- How borrowers can abuse from massive debt repayments with frontrunning attacks to absorb a higher repayment amount (**PIN-13**).

2. Assessment and Scope

The **2PI Network Loans** contracts enable income solutions for borrowers and lenders by providing collateral and liquidity pools where users can provide liquidity and borrow various assets. The protocol brings collateral pools where users deposit assets that serve as collateral to request loans and liquidity pools that allow users to lend and borrow.

The audit started on **Jan 9, 2023** and was conducted on the **audit-2023-01** branch of the git repository at https://github.com/2pinetwork/Loans/ as of commit **251c7d62f3be61ce461d8224fa597648c7228ee2** of **Jan 8, 2023**.

The audited files have the following sha256sum hash:

```
02b0fcd18b2b81c57e84cb6bfaae49da1e4994378026d8a5d8e43a5346b0860f  ./contracts/mocks/MockStrat.sol
f428decad4cf5408f8f4785fd74b8b127956fcb0b40f55edf2246de22477b027  ./contracts/mocks/PriceFeedMock.sol
aec5267c51ce1cb5e4bcdd92f5f1fe35169470b88c47a30ec75fed258885d8b4  ./contracts/mocks/ERC20Mintable.sol
d02d4d6317727091ca4e4404b8edd08f872aae24904e8e28a88398d73e0fe7b6  ./contracts/SafeBox.sol
81bb6b556d92e41b5eddede3077556bbc5f081a030e7326f413c2011ecb60955  ./contracts/PiGlobal.sol
9d341373ea5bb585dd6041e49ba5b3beb1b8701d11c6b63ec16c95454edec229  ./contracts/Controller.sol
8ebe7b3a867eb972cb6889a1f5d3a452f1864f4bc46580095291d7a9bfbade68  ./contracts/DToken.sol
ac76d4fc45fe36c2d4319e5b495b578ab796d4e436c3acb35e27e04dd7b2ec63  ./contracts/CollateralPool.sol
9b713556f7ccbfa37b9481898a5666a8c0c82190fcfcf84c124e8c2f6303bbe8  ./contracts/Oracle.sol
40d6c985242090657b1e6173c68c9cd6d4f29c0bd930cfe3d28d313e384a1cf8  ./contracts/PiAdmin.sol
a3ba3fc93ba841169fd445367d712e2ef443b1bd46ee1d19baab60afdbeae194  ./contracts/DebtSettler.sol
1947a087327583e7de4b221c2a65bd3a042a3f79aa764f5aaa1489afc4cb7386  ./contracts/LToken.sol
b6477a99c7b0d390d856619a769df86ee11345804a8462c42c8f65c65c618d75  ./contracts/LiquidityPool.sol
0b703d89fb4173b783e4897e1e8bdf564373d515d7c7f957882a44b5252d938d  ./libraries/Errors.sol
a11da752555f38a9546cd2e6cb18548d0cd9f1a6ebdb7f2222d9400d3c6ceb1c  ./interfaces/IOracle.sol
ba98d59aaacd18fefc5fabac0fd0c2de401785f53a746e37446a34db3dddfe3d  ./interfaces/IPool.sol
dda71a317199ed7c816a2f1427de008ed1605c36238956835cb05ffe293f6258  ./interfaces/IStrategy.sol
4bb862af9fa90fc6e3939feeb44f2af3707b9972ff827a2477ed2e5adb70c979  ./interfaces/IDebtSettler.sol
39acfd59e052de2c3925fbadf75573189e52a27e90b239c5eea1c7a6b261618e  ./interfaces/IChainLink.sol
61192439b4c6725b86da6a0af3ab878fa45190805ba2aa548c2f84eacb151198  ./interfaces/IPiGlobal.sol
6b66b725d958dd737677a86ed975bb8c9e6f177c5d0b3de21b4e344cc4e9f41f  ./interfaces/IController.sol
```

The project is specified to be compiled using Solidity compiler version 0.8.17 or greater. Over 172 tests were made achieving an average coverage of 95%, that ran smoothly using the instructions in the repository. The codebase was understandable and easy to read.

The following concerns were identified during the assessment of this project.

Coinspect identified that the protocol has a high degree of centralization that allows the administrator to pause pools, withdrawals, deposits, modify key protocol's

parameters and even remove active liquidity and collateral pools considerably harming users. It is suggested to reduce the degree of centralization by implementing timelocks, separating privileges, and clearly documenting which rights are assigned to each role.

The collateral pool works as an entry point allowing users to deposit and withdraw assets, enabling them to work as collateral for future borrows. In the event of having a borrow position in an expired liquidity pool or having an unhealthy position, they can be liquidated. However, Coinspect identified two scenarios that lead to unprofitable liquidations: expensive gas network conditions (**PIN-11**) and abuse of a rounding error while getting the liquidable amounts (**PIN-21**). Moreover, the account with admin privileges is able to drain the base asset of a pool via the recovery function (**PIN-16**).

Each collateral pool has a Controller contract that handles deposits, withdrawals and liquidations among other key operations. Upon withdrawal, users' shares are burned receiving the counterpart in assets. However, there are some cases where users might get all their withdrawn shares burned without receiving the right asset counterpart (**PIN-10**). The Controller owner is able to set investing strategies to reinvest assets. Then, the controller transfers assets back and forth to the active strategy triggered by withdrawals, repayments and deposits without taking into account that strategies might incur losses (**PIN-12**). The mentioned token transfers are handled by a hook that sends assets to the current strategy, which could be inefficient for small deposit amounts (**PIN-22**). ***Coinspect strongly recommends auditing controllers along with strategies' implementations as they are deeply related to guarantee that security properties are tightly coupled.***

Liquidity pools allow users to provide liquidity or borrow assets if they have previously deposited collateral to the protocol. On deployment, each liquidity pool deploys its liquidity token. More than one liquidity pool can be deployed for the same asset with different due dates and consequently will use liquidity tokens with the same metadata (**PIN-19**). In addition, Coinspect identified that the first depositor is able to steal other's assets with a front-running attack (**PIN-14**). Also, users are allowed to massively repay other's debt via liquidity pools providing the repayment assets. This feature is also open to a front-running attack where a borrower is able to absorb most of the assets used to repay a global debt (**PIN-13**).

In addition, massive repayments will revert for pools having more than 350 borrowers (**PIN-20**). Interest tokens are also handled by each liquidity pool. Their handling and minting design does not consider an interest rate increase that will unfairly mint users a higher amount of interest tokens than expected (**PIN-18**). In relationship with the interest tokens handling, users can calculate a borrow amount that does not mint interest tokens to avoid paying pool's fees while repaying the debt (**PIN-23**).

The protocol's pools are expected to be deployed to work with different assets, providing users with different investment alternatives, nevertheless, using fee-on-transfer tokens will break main protocol's functionality as they are not supported (**PIN-15**).

An oracle that consumes multiple data feeds allows the protocol to calculate the debt and collateral amount expressed in USD. As users are allowed to provide collateral and borrow, the calculations mentioned before are made for each registered pool. Coinspect identified that a stale oracle will revert while performing those calculations, potentially bricking sensitive transactions such as liquidations (**PIN-17**).

3. Summary of Findings

Id	Title	Total Risk	Fixed
PIN-10	Users withdrawals can receive less assets than expected	High	✔
PIN-11	Unprofitable liquidations	High	!
PIN-12	Controller assumes that strategies will always profit	High	!
PIN-13	Frontrunning massive repays benefit big borrows	High	✔
PIN-14	First depositor can steal subsequent depositors' assets	Medium	✔
PIN-15	Fee-on-transfer assets break pools functionality	Medium	!
PIN-16	Admin role can drain pool's collaterals at will	Low	✔
PIN-17	Stale Oracle prevents unrelated user's liquidations	Medium	✔
PIN-18	Unfair interest accrual when rate is modified	Medium	✔
PIN-19	Different liquidity tokens with identical metadata	Medium	✔
PIN-20	Massive debt building reverting due to unbounded loop	Medium	✔
PIN-21	Rounding error leads to unprofitable liquidations	Low	✔
PIN-22	Strategy hooks in controller will reduce profitability of operations	Low	!
PIN-23	Debt repayment fees bypass	Low	✔

PIN-24	Logic error in comparisons with unsigned integers	Info	✔
PIN-25	Misleading comment on withdrawal event	Info	✔
PIN-26	Incorrect usage of EnumerableSet while registering liquidity pools	Info	✔
PIN-27	Almost expired liquidity pools deployments allowed	Info	✔
PIN-28	Users can’t preview deposits or withdrawals results	Info	✔
PIN-29	Over centralized protocol	Info	✔

4. Detailed Findings

PIN-10 Users withdrawals can receive less assets than expected

Total Risk	Impact	Location
High	High	`Controller.sol`
Fixed ✔	Likelihood High	

Description

During withdrawals, users could receive less assets than expected. At the same time, all the user provided shares will be burned. This happens if there is an active strategy and the current balance is not enough to match the expected amount.

The `_withdraw` function first burns all the shares but transfers only a portion of the tokens under the conditions mentioned before:

```
      function _withdraw(address _senderUser, uint _shares, uint _amount, bool _withFee)
internal returns (uint) {
        if (_amount <= 0 || _shares <= 0) revert Errors.ZeroAmount();

        _burn(_senderUser, _shares);

        uint _balance = assetBalance();

        if (_balance < _amount) {
            if (! _withStrat()) revert InsufficientBalance();

            uint _diff = _amount - _balance;

            // withdraw will revert if anyything weird happend with the
            // transfer back but just in case we ensure that the withdraw is
            // positive
            uint withdrawn = strategy.withdraw(_diff);
            if (withdrawn <= 0) revert CouldNotWithdrawFromStrategy();

            _balance = assetBalance();
            if (_balance < _amount) _amount = _balance;
        }
        {...}
            asset.safeTransfer(pool, _amount);
    }
```

With an active strategy and a smaller available balance than the intended amount to be withdrawn, the asset `_amount` is replaced by the smaller strategy balance (`_balance`).

The burned shares are higher than the received counterpart, essentially underpricing the shares.

Recommendation

In case of not having enough balance to cover a withdrawal, burn only the amount of shares that can be actually backed.

Status

Partially Fixed.

On commit `adf4409df21964093ff9df716d69f39fcab1cecf` it is now taken into account the effective available balance to calculate the amount of shares to burn. However, Coinspect identified that this process introduces a cross-contract reentrancy that could happen if a hookable asset is used. The implementation transfers the fees before burning the shares of the user:

```
        // In case the strategy returns less than the requested amount
        if (_toTransfer < _amount) _shares = _toTransfer * _shares / _amount;

        if (_withFee) {
            uint _withdrawalFee = _toTransfer * withdrawFee / RATIO_PRECISION;

            if (_withdrawalFee > 0) {
                _toTransfer -= _withdrawalFee;

                asset.safeTransfer(treasury, _withdrawalFee);
                emit WithdrawalFee(_withdrawalFee);
            }
        }

        _burn(_senderUser, _shares);

        asset.safeTransfer(pool, _toTransfer);

        _strategyDeposit();

        return (_toTransfer, _shares);
```

If the asset has logic that re-enters a different pool in the `_beforeTransfer()` hook, the user who is withdrawing could be able to interact with other pools of the protocol and still have the share balance that has not been burned yet. Perform asset transfers after burning the user's shares.

Fixed on commit `c29e426027b8dd4bb447c027d17082b3c83f3deb`.

PIN-11	Unprofitable liquidations		
Total Risk **High**	Impact High	Location `CollateralPool.sol`	
Fixed !	Likelihood High		

Description

The gas fees spent to call `collateralPool.liquidationCall()` could outweigh the underlying value of the claimed liquidated assets, making the process unprofitable for liquidators.

Each liquidation call consumes between 270,000 (for an expired pool liquidation) and 360,000 (for an unhealthy position liquidation, having only one collateral and liquidity pool registered) gas units. The latter increases if the protocol decides to include more pools. Considering a conservative gas cost of 20 Gwei for a congested network, an ETH price of 1,300 USD and a consumption of 360,000 gas units:

```
360,000 * 20 gwei = 0.0072 ETH = 9.36 USD
```

In the event of having a total liquidatable position (`current position + liquidationBonus`) whose underlying USD value is lower than the cost mentioned before, the liquidation won't be profitable and is unlikely to happen, accumulating bad debt during this period. Also, adding more available pools to the protocol increases the gas spent to calculate the available collateral and liquidable amounts.

It is worth mentioning that a bad debt of 1.7MM USD was generated on Aave last year.

Recommendation

Enforce a minimum borrow position size in order to incentivize liquidations even in congested network conditions.

Status

Won’t Fix.

The 2PI team stated that if the case illustrated in this issue happens during a high gas price scenario, they will face the cost of liquidating those positions.

PIN-12 Controller assumes that strategies will always profit

Total Risk	Impact	Location
High	High	`Controller.sol`
Fixed	Likelihood	
!	High	

Description

Strategies associated with the collateral pool's controller reinvest tokens expecting to get a return on investment. However, the Controller's implementation assumes that strategies will always get positive returns on investment. In the event of having a strategy that incurs in losses, liquidations and key collateral pool's functionalities will fail.

The token flows of collateral pools are managed by the `Controller` contract. The owner is allowed to add strategies that intend to generate an additional yield. The market price of shares or available assets by the strategy might fluctuate according to the outcome of investment, as shown in the example below.

The strategy opens a short position in ETH, giving in exchange some assets. Later, the ETH price rises and the strategy incurs losses. When the strategy closes the short position, it receives less assets than the deposited amount. As a result, the available assets in the strategy decreased.

Under that scenario, users might start withdrawing their positions through the controller. Those whose positions are big enough (or that took too long to withdraw their position), might not be able to claim their assets back:

```
    function _withdraw(address _senderUser, uint _shares, uint _amount, bool _withFee) internal returns
(uint) {
        if (_amount <= 0 || _shares <= 0) revert Errors.ZeroAmount();

        _burn(_senderUser, _shares);

        uint _balance = assetBalance();

        if (_balance < _amount) {
            if (! _withStrat()) revert InsufficientBalance();

            uint _diff = _amount - _balance;

            // withdraw will revert if anyything weird happend with the
```

```
            // transfer back but just in case we ensure that the withdraw is
            // positive
             uint withdrawn = strategy.withdraw(_diff);
             if (withdrawn <= 0) revert CouldNotWithdrawFromStrategy();
        {...}
}
```

In addition, as liquidations call the mentioned function they might also revert under the scenario mentioned before.

Recommendation

Strategies and their interactions with third party protocols should be evaluated and reviewed. Clearly document the expected interactions, outcomes, risks and impacts on Collateral Pools associated with each Strategy. At the moment of this audit, strategies were not available. Coinspect suggests auditing the controller along with its strategies.

Status

Won't Fix.

The 2PI team stated that this issue will be clarified in a future audit for the strategies' implementations.

PIN-13 Frontrunning massive repays benefit big borrows

Total Risk	Impact	Location
High	High	`LiquidityPool.sol` `DebtSettler.sol`
Fixed ✔	**Likelihood** High	

Description

A massive debt repayment can be triggered by calling `buildMassiveRepay()`. Users willing to borrow can front run the mentioned function call reducing their debt in a higher proportion than others', absorbing most of the tokens supplied for the repayments.

Massive repayments require the massive repayer to send the amount of assets used to terminate current debts:

```
 function buildMassiveRepay(uint _amount) external nonReentrant {
    asset.safeTransferFrom(msg.sender, address(debtSettler), _amount);
    debtSettler.build(_amount);
}
```

The `debtSettler.build()` function spreads proportionally the `_amount` of tokens across the current accumulated debt:

```
function build(uint _amount) external onlyPool nonReentrant {
    {...}
    for (uint i = 0; i < _length; i++) {
        address _borrower = _borrowers.at(i);
        uint _credit = _amount * _debts[i] / _totalDebt;
        (, uint _currentCredit) = _records.tryGet(_borrower);

        _credit += _currentCredit; // we have to accumulate each time =)

        _records.set(_borrower, _credit);
    }
}
```

Then, a subsequent external call to `debtSettler.pay()` is required to process the debt repayments which will use the `_credit` amount awarded to each user.
A user willing to borrow can front run a `buildMassiveRepay()` call, abusing the credit calculation to get a higher proportion as they increase their `_debt[i]`, modifying others' debt repayment amount.

Assuming that the massive repayer calls `buildMassiveRepay(currentGlobalDebt)` and a borrow is mined before that call, the following scenarios will happen:

```
FRONT RUNNING BUILDMASSIVEREPAY
 Initial States
  Alice's Debt: 100000000000000000
  Bob's   Debt: 0

 During Frontrunning States
  Alice's Debt: 100000000285388127
  Bob's   Debt: 10000000019025875100

 Final States
  Alice's Debt: 99009901462929590
  Bob's   Debt: 9900990136874208865

FRONT RUNNING DISABLED
 Initial States
  Alice's Debt: 100000000000000000
  Bob's   Debt: 0

 Final States
  Alice's Debt: 190258752
  Bob's   Debt: 0
```

This output shows two different scenarios. The first one considers a front-running borrow call to `buildMassiveRepay()` and the other one represents an expected (non attacked) scenario. It can be seen how the attacker takes a higher proportion of the repayment amount provided.

Recommendation

Use a time-weighted mechanism to massively repay older borrow positions before paying newer ones.

Status

Fixed.

Coinspect reviewed commit `3b5beef31d1d1eb7e11d1e74a017a3a9b5fdd998` which does not address the root cause of this issue. Fixed on commit `f1ad8dee558d8333e077ec6191012ad7f246823c` by adding a time weighted mechanism.

PIN-14 First depositor can steal subsequent depositors' assets

Total Risk	Impact	Location
Medium	High	`LiquidityPool.sol`
Fixed ✔	Likelihood: Medium	

Description

The first depositor of a pool can be victim of a front-running attack and receive less shares than expected due to a rounding error on the shares calculation.

Because the amount of LTokens minted when there's no supply equals the assets provided, an attacker can leverage this calculation and considerably reduce the amount of shares that the victim receives once their deposit is processed.

The `_deposit` implementation calculates the amount of shares as it follows depending on the current LToken's supply:

```
if (_supply <= 0) {
     _shares = _amount;
 } else {
     _shares = (_amount * _supply) / _before;
```

Also, the `_before` variable is calculated with the `_balanceForSharesCalc` function:

```
function _balanceForSharesCalc() internal view returns (uint) {
     return balance() + dToken.totalSupply() + _safeBalance();
}
```

A malicious user can steal a portion of the first depositor's amount by conducting the following steps:

- Bob wants to deposit 2,000 tokens to the pool.

- Alice was scanning the mempool and called deposit(1) minting one share backed by one token. Afterwards, she sends 1,000 tokens to the pool by a regular transfer.
- When Bob's transaction is mined, the shares he gets are rounded down and only **one** share will be minted. This is because `[2,000 * supply / (1,001) = 1.99]`.
- Alice withdraws her share and gets `3,001 * 1 / 2 = 1500 tokens`, making a `500` token profit.
- Bob withdraws and gets `1500` tokens, incurring a `500` token loss.

Proof Of Concept:

```
    it('Should steal others deposit', async function () {
      const { alice, bob, lPool, lToken, token } = await loadFixture(deploy)

      // Alice will be the attacker and Bob the victim.
      await token.mint(alice.address, 1_001)
      await token.mint(bob.address, 2_000)
      await token.connect(alice).approve(lPool.address, 1_001)
      await token.connect(bob).approve(lPool.address, 2_000)

      console.log('\nAlice Balance Before Attack')
      console.log(`LToken: ${await lToken.balanceOf(alice.address)}`)
      console.log(`Token: ${await token.balanceOf(alice.address)}`)

      console.log('\nBob Balance Before Attack')
      console.log(`LToken: ${await lToken.balanceOf(bob.address)}`)
      console.log(`Token: ${await token.balanceOf(bob.address)}`)

      // Bob wants to deposit 2_000e18 tokens. Alice frontruns him.
      expect(await lPool.connect(alice)['deposit(uint256)'](1)).to.emit(lPool, 'Deposit')
      console.log('\nAlice Received Shares')
      console.log(`LToken: ${await lToken.balanceOf(alice.address)}`)

      // Alice sends 1_000e18 tokens to the contract.
      await token.connect(alice).transfer(lPool.address, 1_000)

      // Bob tx is mined
      expect(await lPool.connect(bob)['deposit(uint256)'](2_000)).to.emit(lPool, 'Deposit')
      console.log('\nBob Received Shares')
      console.log(`LToken: ${await lToken.balanceOf(bob.address)}`)

      // Alice withdraws her repriced share
      expect(await lPool.connect(alice)['withdraw(uint256)'](1)).to.emit(lPool, 'Withdraw')

      console.log('\nAlice Balance After Attack')
      console.log(`LToken: ${await lToken.balanceOf(alice.address)}`)
      console.log(`Token: ${await token.balanceOf(alice.address)}`)

      console.log('\nBob Balance After Attack')
      console.log(`LToken: ${await lToken.balanceOf(bob.address)}`)
      console.log(`Token: ${await token.balanceOf(bob.address)}`)

      // Bob Withdraws his shares
      expect(await                              lPool.connect(bob)['withdraw(uint256)'](await
lToken.balanceOf(bob.address))).to.emit(lPool, 'Withdraw')
```

```
    console.log('\nBob Balance After Withdrawing')
    console.log(`LToken: ${await lToken.balanceOf(bob.address)}`)
    console.log(`Token: ${await token.balanceOf(bob.address)}`)
  })
```

Recommendation

Require the first depositor to mint a minimum amount of shares. Also, a small portion could be sent to a non-existing address and the first depositor should receive (initialAmount - BURN_AMOUNT).

Status

Fixed on commit `8938b6d44d113a95f228d95bf355228a57d5c41e`.

A minimum supply of `10**3` liquidity tokens were added. The exploit illustrated in this finding was run again and reverted with the `LowSupply()` error triggered when the total supply of liquidity tokens after depositing is below the minimum supply.

PIN-15	Fee-on-transfer assets break pools functionality		

Total Risk
Medium
	Impact
	High
		Location
		`LiquidityPool.sol`
		`CollateralPool.sol`	

Fixed !	Likelihood Low		

Description

Using tokens that charge a fee on transfer will make pools behave unexpectedly, potentially breaking their functionality.

Coinspect identified four scenarios that harm users and the protocol if fee on transfer tokens are used.

First Scenario: Borrowed amount imbalance

While borrowing an asset from the pool, the amount of debt tokens minted matches the input of `borrow()`:

```
function borrow(uint _amount) external nonReentrant whenNotPaused notExpired {

    { … }

     // Mint interest tokens
     iToken.mint(msg.sender, _interestTokens);
     // Mint real debt tokens
     dToken.mint(msg.sender, _amount);

     // Set last interaction
     _timestamps[msg.sender] = uint40(block.timestamp);

     asset.safeTransfer(msg.sender, _amount);

     if (_hasDebtSettler()) debtSettler.addBorrower(msg.sender);

     emit Borrow(msg.sender, _amount);
 }
```

As a result, users incur in a debt greater than the effective amount received (proportional to the current fee on transfer).

Second Scenario: Shares value leak

Users will receive more shares than the effective amount transferred to the pool while depositing. This is because the amount is calculated according to the input provided instead of using the actual amount transferred (`_amount - feeOnTransfer`).

```
function _deposit(uint _amount, address _onBehalfOf) internal {
     uint _before = _balanceForSharesCalc();

     asset.safeTransferFrom(msg.sender, address(this), _amount);

     // SaveGas
     uint _supply = lToken.totalSupply();
     uint _shares;

     if (_supply <= 0) {
         _shares = _amount;
     } else {
         _shares = (_amount * _supply) / _before;
     }

     if (_shares <= 0) revert Errors.ZeroShares();

     lToken.mint(_onBehalfOf, _shares);

     emit Deposit(msg.sender, _onBehalfOf, _amount, _shares);
 }
```

Third Scenario: Withdrawal Brick

While withdrawing from a `LiquidityPool`, the required amount will never be reached because the fee on transfer will cause the contract balance to asymptotically approach the required balance, reverting each `withdraw()` (as the token transfer amount will exceed current balance). This will happen in case of requesting funds from the `SafeBox` (when the balance of the asset is less than the withdrawn amount).

```
function _withdraw(uint _shares, address _to) internal returns (uint) {
     {...}

     // Ensure if we don't have the entire amount take it from safe
     if (_amount > _assetBal) safeBox.transfer(_amount - _assetBal);

     asset.safeTransfer(_to, _amount);

     {...}
 }
```

Fourth Scenario: Repayment Brick

Liquidations (through `_repay()`) transfer assets from the repayer to the pool contract. In the event of using a fee on transfer token, the effective amount repaid will be smaller than the actual repayment amount. If fees are collected, the liquidation call might revert if no tokens are initially in the contract.

```
function _repay(address _payer, address _account, uint _amount) internal {
     {...}
     // Take the payment
     asset.safeTransferFrom(_payer, address(this), _amount);

     uint _piFee = 0;
     // charge fees from payment
     if (_interestToBePaid > 0) _piFee = _chargeFees(_interestToBePaid);

     // Send the payment to safeBox
     if (safeBoxEnabled) asset.safeTransfer(address(safeBox), _amount - _piFee);

     emit Repay(_account, _amount);
 }
```

Recommendation

Clearly document that the protocol does not support fee-on-transfer tokens. If fee-on-transfer tokens will be used (e.g. USDT is a fee-on-transfer token with the fee set to zero), redesign the main pool's functionalities so their handling is made with the amount of tokens before and after transfers rather than using fixed amounts.

Status

Acknowledged.

The 2PI team stated: *“Will not support fee-on-transfer tokens, except for USDT (until they put a >0 fee)”*.

PIN-16	Admin role can drain pool's collaterals at will		
Total Risk **Low**	Impact Low	Location `CollateralPool.sol`	
Fixed ✔	Likelihood Low		

Description

The account with the `admin` role has the privilege to rescue mistakenly sent funds to a collateral pool. However, the recovery function allows that account to retrieve not only arbitrary assets but also the pool's base asset.

```
function rescueFounds(IERC20Metadata _asset) external nonReentrant onlyAdmin {
    address _treasury = piGlobal.treasury();

    _asset.safeTransfer(_treasury, _asset.balanceOf(address(this)));
}
```

If the account with the admin role is compromised, each pool can be easily drained by passing the pool's base asset address in `rescueFounds()`.

Recommendation

Check that the input `_asset` is not the pool's base asset. Modify `'rescueFounds'` for `'rescueFunds'`.

Status

Not an issue under current conditions.

The team clarified that the Collateral pool won't hold any funds as they flow directly to the Controller. However, this issue will be substantial if funds end up at sometime in the Collateral pool.

PIN-17 Stale Oracle prevents unrelated user's liquidations

Total Risk	Impact	Location
Medium	High	`Oracle.sol`
Fixed ✔	Likelihood: Medium	

Description

The collateral and debt calculations require getting the asset's prices through oracles. However, as those calculations are performed by looping over all the registered assets and pools, a user that has never interacted with a certain pool (e.g. by opening a position) might not be able to get liquidated if the underlying oracle is failing.

The debt and collateral calculations are performed by looping over each liquidity and collateral pool getting the current underlying asset's price:

```
function _borrowedInUSD(address _account) internal view returns (uint _amount) {
    address[] memory _lPools = piGlobal.liquidityPools();

    for (uint i = 0; i < _lPools.length; i++) {
        ILPool _pool = ILPool(_lPools[i]);
        uint _price = _normalizedPrice(_pool.asset());
    {...}
}

function _collateralInUSD(address _account) internal view returns (uint _availableInUSD) {
    address[] memory _cPools = piGlobal.collateralPools();

    for (uint i = 0; i < _cPools.length; i++) {
        ICPool _pool = ICPool(_cPools[i]);
        uint _price = _normalizedPrice(_pool.asset());
    {...}
}
```

Each Oracle's price is retrieved with the `_normalizedPrice` function that reverts upon invalid prices or liveliness:

```
   function _normalizedPrice(address _asset) internal view returns (uint) {
    (
        uint80 _id,
        int _roundPrice,
        ,
        uint _timestamp,
        uint80 _answeredInRound
    ) =  priceFeeds[_asset].latestRoundData();
    if (_id < _answeredInRound) revert OldPrice();
```

```
        if (_timestamp < block.timestamp - priceTimeToleration) revert OldPrice();
      {...}
  }
```

The following scenario might happen:

- Alice deposits in the `CollateralA` pool and borrows `tokensB` from the `LiquidityB` pool.
- The `tokenC's` oracle is down.
- The price of `tokensB` drops and her position is unhealthy. Alice can't be liquidated because the `oracle.getLiquidableAmounts()` call inside `liquidationCall()` reverts.

Recommendation

Consider only pools where a user deposited or borrowed for debt and collateral calculations.

Status

Fixed in commit `f071647b88bbef2b8026f24730954ebde90678f8`.

The mentioned functions above now only calls `_normalizedPrice()` for the pools where the user has collateral and debt.

PIN-18 Unfair interest accrual when rate is modified

Total Risk	Impact	Location
Medium	High	`LiquidityPool.sol`
Fixed ✔	Likelihood: Medium	

Description

Users will pay unexpected interests for their debts if the rate is modified after the position creation.

Each liquidity pool contract tracks the amount of debt and interest for each user by minting debt and interest tokens. The last ones increase as the time passes representing the accumulated interest by the debt position. Those tokens are only minted/deducted when a user borrows more or repays his debt. If the owner decides to modify the interest rate, all the unaccrued interest tokens (`noMintedInterest`) will be minted with the newer interest rate.

The root of this is how unaccrued interest tokens are handled:

```
function _debtTokenDiff(address _account) internal view returns (uint) {
    uint _bal = dToken.balanceOf(_account);

    if (_bal <= 0 || _timestamps[_account] <= 0) return 0;

    // Difference between the last interaction and (now or due date)
    uint _timeDiff = block.timestamp;

    if (_timeDiff > dueDate) _timeDiff = dueDate;

    _timeDiff -= _timestamps[_account];

    // Interest is only calculated over the original borrow amount
    // Use all the operations here to prevent _losing_ precision
    return _bal * (interestRate + piFee) * _timeDiff / SECONDS_PER_YEAR / PRECISION;
}
```

The mentioned function is called by `debt()` that calculates the debt for a user in that liquidity pool:

```
function _debt(address _account) internal view returns (uint, uint, uint, uint) {
    uint _dBal = dToken.balanceOf(_account);
    uint _iBal = iToken.balanceOf(_account);

    if (_dBal <= 0 && _iBal <= 0) return (0, 0, 0, 0);

    uint _notMintedInterest = _debtTokenDiff(_account);

    // Interest is only calculated over the original borrow amount
    return (_dBal, _iBal, _notMintedInterest, _dBal + _iBal + _notMintedInterest);
}
```

The following scenario will happen if the interest rate is modified:

- Alice opens a borrow position and has a balance of `100 dTokens` (representing her debt).
- A few periods later (e.g. two weeks), her position is still healthy and she has never called `borrow()` again or `repay()`.
- Suddenly the owner decides to increase the interest rate.

As a result, the overall debt of Alice increased as if she requested a borrow position at the new interest rate, instead of considering that the `noMintedInterests` correspond to unaccrued interests for a lower rate. Interests for a specific period should be accrued with the interest rate of that period.

Recommendation

Timelock the interest rate setter. Clearly document this behavior, mentioning that interest tokens will be accrued when borrowing or repaying.

Status

Fixed on commit `88202b59481cf3000ba6d30e805ae4994052c421`.

As this process was only reproducible if the initial fee was set to zero (and then increased), pools are now paused by default on deployment allowing the admin to fully configure it before going operational.

PIN-19 Different liquidity tokens with identical metadata

Total Risk	Impact	Location
Medium	High	`LiquidityPool.sol` `LToken.sol`
Fixed ✔	Likelihood Low	

Description

Users could confuse tokens with different maturities as they all have the same symbol and name.

Upon deployment, each liquidity pool deploys on its constructor liquidity tokens. Because the only parameter used to differentiate each liquidity token is the asset's name and symbol, pools with the same asset but different due date will deploy two different tokens with the same metadata. Users might be unaware of this and receive or send tokens of an already expired pool.

The constructor of the liquidity tokens assigns their name and symbol with the following structure:

```
constructor(IERC20Metadata _asset) ERC20(
    string(abi.encodePacked("2pi Liquidity ", _asset.symbol())),
    string(abi.encodePacked("2pi-L-", _asset.symbol()))
) {
    asset = _asset;
    pool = msg.sender;
}
```

Then, each liquidity pool instantiates a new liquidity token upon deployment, passing the asset's contract:

```
constructor(IPiGlobal _piGlobal, IERC20Metadata _asset, uint _dueDate) {
     {...}
    asset = _asset;
    dueDate = _dueDate;

    // Liquidity token
    lToken = new LToken(asset);
    // Debt token
    dToken = new DToken(asset);
    // Interest token
    iToken = new DToken(asset);
```

```
        treasury = _piGlobal.treasury();
        piGlobal = _piGlobal;
    }
```

Because the liquidity tokens are transferable, users might send, receive and interact with tokens that represent already expired pools. Attackers might leverage from this to scam other users with the following process:

- Mallory has 100 `LTokensA` that belong to an already expired pool
- Alice has 100 `LTokensB`, of an active pool.
- Mallory sends 100 `LTokensA` to Alice and contacts her asking for the tokens back, tricking Alice to transfer the `LTokensB` instead.

Also, Etherscan latest `ERC-20` transactions only show the token's name. To get the address, users have to go to the token's page:

	Txn Hash	Age	From		To	Value	Token
	0x10261598ca850e1295...	16 mins ago	0x4e3c1f7361bc3fdebed...	OUT	0x17c1ae82d993792400...	975.295704	Tether USD (USDT)
	0x02c134aa1bddcfe18f1...	1 day 22 hrs ago	0x4e3c1f7361bc3fdebed...	OUT	0x43d7f734b67d9510d4...	300	Tether USD (USDT)
	0x75d3c640ed5bb13963...	1 day 23 hrs ago	Binance 15	IN	0x4e3c1f7361bc3fdebed...	291.533021	Tether USD (USDT)
	0x9c0d924dbc6f3a7b34...	2 days 6 hrs ago	Binance 17	IN	0x4e3c1f7361bc3fdebed...	13,342.8715	Tether USD (USDT)
	0x858b96442100e3d858...	2 days 9 hrs ago	Binance 15	IN	0x4e3c1f7361bc3fdebed...	56.25	ApeCoin (APE)

Recommendation

Add the liquidity pool's `endDate` to the LToken's metadata.

Status

Fixed on commit `0c68f49a82b30313b1693a365827b2e327349d7b`.

The liquidity tokens now include the due date in their name and symbol.

PIN-20 Massive debt building reverting due to unbounded loop

Total Risk	Impact	Location
Medium	Medium	`LiquidityPool.sol` `DebtSettler.sol`
Fixed ✔	Likelihood Medium	

Description

The `build()` function loops over all the registered borrowers distributing the massive repayment amount across all the borrowers. In the event of having more than 350 borrowers, this function will revert as it might exceed the max. gas.

The mentioned function first calculates the total debt and then, distributes credit across all the active borrowers. Both times looping over every single borrower:

```
function build(uint _amount) external onlyPool nonReentrant {
    uint _totalDebt = 0;
    uint _length = _borrowers.length();

    // Prevent double call to _debt()
    uint[] memory _debts = new uint[](_length);

    // Get the total debt for all borrowers
    for (uint i = 0; i < _length; i++) {
        _debts[i] = pool.debt(_borrowers.at(i));

        _totalDebt += _debts[i];
    }

    if (_totalDebt <= 0) return;
    if (_amount > _totalDebt) _amount = _totalDebt;

    for (uint i = 0; i < _length; i++) {
        address _borrower = _borrowers.at(i);
        uint _credit = _amount * _debts[i] / _totalDebt;
        (, uint _currentCredit) = _records.tryGet(_borrower);

        _credit += _currentCredit; // we have to accumulate each time =)
        _records.set(_borrower, _credit);
    }
}
```

As the amount of borrowers will likely increase with time, the same will happen with the gas used to distribute the credit across debtors and eventually this consumption might exceed the max gas per block.

The following example shows how with 350 active borrowers, the `buildMassiveRepay()` exceeds the maximum gas per block (30MM):

```
[OUTPUT]
  Gas used to build massive repay 10 borrowers:   930,486
  Gas used to build massive repay 50 borrowers:   4,227,026
  Gas used to build massive repay 100 borrowers:  8,347,737
  Gas used to build massive repay 250 borrowers:  20,710,105
  Gas used to build massive repay 350 borrowers:  over 30MM
```

```
  it('increases gas spent while building when there are more borrowers', async function () {
       const fixtures = await loadFixture(deploy)

       const {
       alice,
       bob,
       dToken,
       cPool,
       lPool,
       debtSettler,
       token,
       treasury,
       } = fixtures

       await setupCollateral(fixtures)

       // Add liquidity & Repayment
       await token.mint(lPool.address, 50e18 + '')

       let amountOfBorrowers = 300;
       for (let index = 0; index < amountOfBorrowers; index++) {
        // get a signer
        let curSigner = ethers.Wallet.createRandom();
        // add it to Hardhat Network
        curSigner = curSigner.connect(ethers.provider);
        // send some gas ether
        await alice.sendTransaction({to: curSigner.address, value: ethers.utils.parseEther('0.3')});
        // generate token balance
        await token.mint(curSigner.address, 1e18 + '')
        // approve and deposit
        await token.connect(curSigner).approve(cPool.address, 100e18 + '')
        await expect(cPool.connect(curSigner)['deposit(uint256)'](1e17 + '')).to.emit(cPool, 'Deposit')

        // console.log(`Borrowing for user #${index + 1} with address ${curSigner.address}`)
        await lPool.connect(curSigner).borrow(1);
        let signerDebt = await lPool['debt(address)'](curSigner.address)
        expect(signerDebt).to.be.eq(1);
       }

       await token.mint(treasury.address, 100e18 + '')
       await token.connect(treasury).approve(lPool.address, 100e18 + '')

       let buildTx = await lPool.connect(treasury).buildMassiveRepay(ethers.utils.parseEther('50'))
       let buildReceipt = await buildTx.wait()
       console.log(`\nGas    used    to    build    massive    repay    ${amountOfBorrowers}    borrowers:
${buildReceipt.gasUsed}`)
   })
```

Recommendation

Redesign the massive repayment mechanism.

Status

Fixed by splitting each loop depending on the gas left.

On commit `3b5beef31d1d1eb7e11d1e74a017a3a9b5fdd998` the build and pay mechanism was modified to prevent out of gas scenarios. However, Coinspect identified that the Handler (new role) is not able to specify lower and upper loop indexes. Instead, they are related to the current amount of borrowers and the last index before being out of gas. Two scenarios might arise because of this that need to be addressed:

1) It could be possible to create sybil borrow positions to extend the building period.
2) It is possible to alter the amount of total minted debt across debt and interest tokens to stop the massive payment. An issue describing this behavior is depicted in the following report.

PIN-21 Rounding error leads to unprofitable liquidations

Total Risk	Impact	Location
Low	Low	`Oracle.sol` `CollateralPool.sol`
Fixed ✔	Likelihood Low	

Description

Users can borrow a small amount of assets in order to generate debt which liquidation yields zero tokens to the liquidator because of a rounding issue in `oracle.getLiquidableAmounts()`.

This scenario happens when: 1) the pool has expired and 2) the position's asset is the same as the token received by liquidators. Moreover, users must have a fully collateralized position (that prevents their liquidation because of having a low health factor).

With the environment conditions provided by default in the test suite (default protocol's parameters), any liquidated borrow positions up to 99 tokens (just 99, without decimals) will transfer zero tokens to the liquidator. **This is because the `_liquidableCollateral` calculation rounds down**:

```
      function getLiquidableAmounts(address _account, address _liqPool) external view returns (uint
_liquidableCollateral, uint _debtToBePaid) {
     {...}
       if (_lPool.expired()) {
           // If collateral without bonus in USD is greater or equal to debt we liquidate the entire debt
           if (_availableCollateralInUSD >= _debtInUSD) _debtToBePaid = _debt;
           // in other case we convert the available collateral in USD to the debt tokens
           else _debtToBePaid = _availableCollateralInUSD * BASE_PRECISION / _lPrice;
       } else {
     {...}

       // _debtToBePaid is the debt tokens to be liquidated so we need to convert it to
       // collateral tokens via the prices
       _liquidableCollateral = _debtToBePaid * _lPrice / _cPrice;
       // have to add the liquidation bonus %
       _liquidableCollateral += _liquidableCollateral * liquidationBonus / BASE_PRECISION;
   }
```

Malicious users might abuse this to generate multiple due borrow positions which liquidation is not profitable, accumulating debt in the protocol.

Proof Of Concept:

Bob borrows `99 tokens` from an active pool. Then, when the pool expires Alice liquidates Bob. As a result, Alice receives no tokens in exchange, wasting gas.

```
  [OUTPUT]
   Bob’s debt: 99
   Tokens received by liquidator: 0
```

```
it('accumulates unprofitable debt', async function () {
 const fixtures = await loadFixture(deploy)

 const {
   alice,
   bob,
   cPool,
   cToken,
   piGlobal,
   token,
   LPool,
   DebtSettler,
 } = fixtures

 const dueDate     = (await ethers.provider.getBlock()).timestamp + 20
 const lPool       = await LPool.deploy(piGlobal.address, token.address, dueDate)
 const debtSettler = await DebtSettler.deploy(lPool.address)

 await Promise.all([
   lPool.setDebtSettler(debtSettler.address),
   token.mint(lPool.address, 10e18 + ''),
   piGlobal.addLiquidityPool(lPool.address),
   setupCollateral({...fixtures, lPool}),
 ])
 // Add liquidity & Repayment
 await token.mint(lPool.address, 10e18 + '')
 await token.mint(bob.address, 10e18 + '')

 const balance = await cToken.balanceOf(bob.address)
 const depositAmount = ethers.utils.parseUnits('99', 1)

 // Skip low HF & LF...
 await lPool.connect(bob).borrow(depositAmount.div(10))

 await mine(20)

 const debt = await lPool['debt(address)'](bob.address)
 console.log(`Bob's debt: ${debt}`)


 // Alice doesn't have any tokens before liquidation call
 expect(await token.balanceOf(alice.address)).to.be.equal(0)

 // Approve for repay
 await token.connect(alice).approve(lPool.address, 100e18 + '')
 await cPool.connect(alice).liquidationCall(bob.address, lPool.address, debt)

 console.log(`Tokens received by liquidator: ${await token.balanceOf(alice.address)}`)

 expect(await lPool['debt(address)'](bob.address)).to.be.equal(0)
 expect(await token.balanceOf(alice.address)).to.be.equal(debt.div(100)) // 1% of bonus
 expect(await cToken.balanceOf(bob.address)).to.be.equal(
   balance.sub(debt.add(debt.div(100))) // 1% of bonus
 )
})
```

Recommendation

Ensure that roundings are made in favor of protocol's solvency.

Status

Fixed on `c8f796e3c69d820e8c7ba92561086eac7122c983`.

The liquidable amount calculation now rounds up to prevent the mentioned behavior.

PIN-22	Strategy hooks in controller will reduce profitability of operations

Total Risk	Impact	Location
Low	Low	`Controller.sol`
Fixed	Likelihood	
!	Low	

Description

The Controller implementation can trigger unnecessary unprofitable transactions.

The `_strategyDeposit()` hook is called after each key action of the `Controller` to send the current asset's balance to the active strategy. This repetitive action could increase the gas spent on each call, potentially making some movements of small amounts unprofitable due to gas costs.

The mentioned function is called in `_withraw()`, `deposit()` and `setStrategy()`:

```
function _strategyDeposit() internal {
   if (! _withStrat()) return;
   // If the line before didn't break the flow, strategy is present
   if (strategy.paused()) return;

   uint _amount = assetBalance();

   if (_amount > 0) {
       asset.safeTransfer(address(strategy), _amount);

       strategy.deposit();
   }
}
```

Users can open small positions but the owner can add later a strategy, making their withdrawal (or even liquidation) unprofitable.

Recommendation

Add logic to only deposit to strategy when a minimum amount is available.

Status

Partially fixed on `603cea7fb024c8de240ae24eb22a5166894d5e33`.

The 2PI team added a `minStrategyDepositAmount` global threshold that is checked when performing a strategy deposit. However, Coinspect identified that its default value (`1e18`) could be prohibitive for pools using a valuable asset such as WBTC. It is suggested to set that amount before going operational taking into account each asset's value and decimals. Also, consider calling `setMinStrategyDepositAmount()` when deploying to mitigate this.

PIN-23 Debt repayment fees bypass

Total Risk	Impact	Location
Low	Low	LiquidityPool.sol
Fixed ✔	Likelihood: Low	

Description

Borrowers can calculate a max borrow amount that will not mint interest tokens, preventing the treasury from receiving rewards (later distributed to liquidity providers) because the interest accrual calculation rounds down.

Users are charged with fees when borrowing from a liquidity pool upon repayment. The amount of fees transferred to the treasury is composed by the current minted interest tokens and the non minted interest tokens (unaccrued):

```
function _repay(address _payer, address _account, uint _amount) internal {
    if (_amount <= 0) revert Errors.ZeroAmount();
    if (_timestamps[_account] == 0) revert NoDebt();

    (
        uint _dTokens,
        uint _iTokens,
        uint _diff,
        uint _totalDebt
    ) = _debt(_account);

    // tmp var used to keep track what amount is left to use as payment
    uint _rest = _amount;
    uint _interestToBePaid = 0;

    if (_amount >= _totalDebt) {
        // All debt is repaid
        _amount = _totalDebt;
        _timestamps[_account] = 0;
        _rest = 0;
        _interestToBePaid = _diff + _iTokens;
   {...}
}
```

The amount of `iTokens` and the `_notMintedInterest (_diff)` is handled by the `_debtTokenDiff()` function:

```
function _debtTokenDiff(address _account) internal view returns (uint) {
    {...}
    // Difference between the last interaction and (now or due date)
    uint _timeDiff = block.timestamp;
```

```
        if (_timeDiff > dueDate) _timeDiff = dueDate;

        _timeDiff -= _timestamps[_account];

        // Interest is only calculated over the original borrow amount
        // Use all the operations here to prevent _losing_ precision
        return _bal * (interestRate + piFee) * _timeDiff / SECONDS_PER_YEAR / PRECISION;
    }
```

Users can calculate a `_bal` amount that makes the return of the last function to be zero. Consequently, no interest tokens are minted and the calculated `_notMintedInterest` will be zero as the `_timeDiff` is updated with the current timestamp. For the default deployment values, that max borrow is `262,799.`

Users are able to call subsequently `borrow(262,799)` up to 300 times or call it once with one `borrow(1)` call per block (that updates the timestamp) before an interest token is minted. The main constraints of this process are the interest rates, Pi Fees, and average time between blocks. If the mentioned parameters tend to zero, the max borrow amount to abuse of this rounding issue tends to infinity.

Recommendation

Round up while calculating the token diff.

Status

Fixed on commit `c8f796e3c69d820e8c7ba92561086eac7122c983`.

The token diff calculation now rounds up.

PIN-24	Logic error in comparisons with unsigned integers

Total Risk	Impact	Location
Info	-	-
Fixed ✔	**Likelihood** -	

Description

On several places across many contracts of the project, it is checked that an unsigned integer variable is smaller or equal than zero. This is a logic error as unsigned integers cannot have negative values and therefore cannot be smaller than zero.

Some instances where this happens:

```
if (_shares <= 0) revert Errors.ZeroShares();

if (_amount <= 0) revert Errors.ZeroAmount();

if (_expectedAmount <= 0) revert Errors.ZeroAmount();
```

Recommendation

Only check equality against zero for unsigned integers under the mentioned scenarios.

Status

Fixed on commit `b571b7a7f6b42a6e3aa152cbafbc51791285fcbb`.

PIN-25 Misleading comment on withdrawal event

Total Risk	Impact	Location
Info	-	`Controller.sol`
Fixed ✔	**Likelihood** -	

Description

The comments on the `WithdrawalFee` and `NewWithdrawalFee` events are the same, but the nature of each event differs.

```
/**
 * @dev Emitted when a new withdraw fee is set
 *
 * @param amount The amount of the fee
 */
event WithdrawalFee(uint amount);

/**
 * @dev Emitted when a new withdraw fee is set
 *
 * @param oldFee The old fee
 * @param newFee The new fee
 */
event NewWithdrawFee(uint oldFee, uint newFee);
```

Recommendation

Modify the comments for the `WithdrawalFee` event.

Status

Fixed on commit `d21490c44693ab10d85bea25d31188bd112897da`.

PIN-26 Incorrect usage of EnumerableSet while registering liquidity pools

Total Risk	Impact	Location
Info	-	`PiGlobal.sol`
Fixed ✔	Likelihood -	

Description

The `addLiquidityPool` calls twice to the add function. First, it is called to check if the value was previously assigned and then called again. The following scenario will happen for a new liquidity pool addresses:

```
    function addLiquidityPool(address _pool) external onlyAdmin nonReentrant {
        if (_pool == address(0)) revert Errors.ZeroAddress();

        if (! liquidityPoolsSet.add(_pool)) revert AlreadyExists();

        liquidityPoolsSet.add(_pool);

        emit NewLiquidityPool(_pool);
    }
```

```
1) addLiquidityPool(somePool)
      a) liquidityPoolsSet.add(somePool) -> true (was successfully
         added, meaning that was not added before)
      b) liquidityPoolsSet.add(somePool) -> false (uncaught return)
      c) Emit NewLiquidityPool(somePool)
```

```
[EnumerableSet.sol]

    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }
```

It is worth mentioning that the implementation of `addCollateralPool` does not include the duplicated `add` instruction.

Recommendation

Fix the `EnumerableSet` usage in the mentioned function.

Status

Fixed on commit `95658babe23a7aae7758eb2ac7c6ef473218ae91`.

PIN-27	Almost expired liquidity pools deployments allowed		
Total Risk Info	Impact -	Location `LiquidityPool.sol`	
Fixed ✔	Likelihood -		

Description

The owner is able to deploy an almost expired liquidity pool by setting an `endDate` a few seconds after the current timestamp:

```
constructor(IPiGlobal _piGlobal, IERC20Metadata _asset, uint _dueDate) {
     if (_dueDate <= block.timestamp) revert DueDateInThePast();
   {...}
}
```

Essentially, this enables the owner to deploy an expired pool allowing only interactions that require from an active pool within that same block.

Recommendation

Evaluate including a minimum pool duration.

Status

Fixed on `31b2838cf11d753f8e84bbfecd37f9ba9a887de6`.

A minimum of 5 days is now enforced. Consider evaluating if pools of lower duration are meant to be deployed in the future (say 1 day) to determine if a 5 day requirement is too restrictive for the expected use cases.

PIN-28 Users can’t preview deposits or withdrawals results

Total Risk	Impact	Location
Info	-	`LiquidityPool.sol` `CollateralPool.sol`
Fixed ✔	Likelihood -	

Description

Users can’t get the amount of shares or assets they will receive after an operation because there are no view methods using the current protocol’s and pool’s states that perform those calculations.

Because of this, users are forced to perform those calculations externally (prone to errors) or interact directly with write functions of the contracts. Moreover, off-chain services won’t be able to get the amounts to be received after a deposit or withdrawal.

The `EIP4626` includes view methods that allow: “on-chain or off-chain users to simulate the effects of their deposits, withdraws, mints, etc, at the current block, given current on-chain conditions”.

Recommendation

Include the mentioned view methods to improve interoperability.

Status

Fixed on commit `5b1290f0efaca003ea073bd6c1363903bb32b57f`.

PIN-29	Over centralized protocol		
Total Risk **Info**	Impact -	Location *	
Fixed ✔	Likelihood -		

Description

The protocol has a high degree of centralization that could potentially harm users if a private key is compromised.

Each pool can be paused by the user holding the `Admin` role. In addition, key protocol parameters can be adjusted by that same account at any time (fees, rates, etc.).

Recommendation

Timelock key setters to bring predictability. Use different roles for the pauser and admin privileges.

Status

Fixed on 0b584c979500fbecbe5ef06ae2508bc9458f8fb0.

A pauser role was added.

5. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.