2pi Network Loans

Security Report

coinspect

2PI Loans Strategies

Smart Contract Audit

V230308 Prepared for 2PI Network • March 2023

1. Executive Summary

In **March 2023, 2PI Network** engaged Coinspect to perform a source code review of **2PI Loans Strategies**. The objective of the project was to evaluate the security of the new investing strategies that can be added to each 2PI pool's controller to enable alternative yield opportunities.

Coinspect highly recommends conducting a new audit to perform a more thorough assessment of the security of those contracts and the integration with the other parts of the protocol. The third-party protocols that these strategies interact with were not in scope.

The following issues were identified during the initial assessment:

High Risk	Medium Risk	Low Risk
Open **0**	Open **0**	Open **0**
Fixed 1	Fixed 3	Fixed 3
Reported 1	Reported 4	Reported 3

Coinspect identified one high-risk issue, three medium-risk issues and three low-risk issues. The high risk issue shows how a non implemented function will prevent a strategy from charging fees and transferring harvested rewards, **PIN-30**. The three medium risk issues illustrate a denial of service when massively repaying debt **(PIN-31)**, how several strategies could be broken because of function implementations with no logic on an abstract **(PIN-32)** and a potential pool drain via a read-only-reentrancy if specific conditions are met **(PIN-33)**.

The first low risk issue, **PIN-34**, illustrates a potential underflow triggered when calculating the prices of each strategy's shares, **PIN-35** shows how rewards might

be unclaimed when migrating a strategy and **PIN-36** depicts how deposits could revert if the amount is below a threshold.

Two informational issues were reported related to how the modification of the performance fee affects previously unharvested rewards and a suggestion to include complete NatSpec on each contract.

2. Assessment and Scope

The audit started on **February 27, 2023** and was conducted on the **audit-2023-02-27 tag** of the **origin/master** branch of the git repository at https://github.com/2pinetwork/Loans as of commit **f7702cafbba9d94f563285f623fc8ba068c3b70d** of **February 26, 2023**.

The audited files have the following sha256sum hash:

```
58ded3f7943957a159e6f0037c238f1d5a2e4cf3662050b7a974a93c652d7dfb  ./contracts/mocks/MockStrat.sol
f428decad4cf5408f8f4785fd74b8b127956fcb0b40f55edf2246de22477b027  ./contracts/mocks/PriceFeedMock.sol
6913c7c0afca759026c63291f3ac94a827c03325bc1f1aaddae0ba8b970e4b34  ./contracts/mocks/ContractInteractionMock.sol
0698e3e55f6898c1989d18bb0103ebe510019721f31465688c954f045e8ffb78  ./contracts/mocks/PiPriceOracle.sol
aec5267c51ce1cb5e4bcdd92f5f1fe35169470b88c47a30ec75fed258885d8b4  ./contracts/mocks/ERC20Mintable.sol
c5fa07a3c6f9e877e2c46c7a4a666800edabdb4e6e6ead754b1ad5ff8233d8c8  ./contracts/JarvisStrat.sol
d02d4d6317727091ca4e4404b8edd08f872aae24904e8e28a88398d73e0fe7b6  ./contracts/SafeBox.sol
64a0823895be2401e86cc953c4bdfcf4092ea8b84f778ca80056bc9682e858f4  ./contracts/BalancerV2Strat.sol
091b5c1af06bbe1c1dc1669077768d33a2a07f1b5b10859a33af651ec340b238  ./contracts/Swappable.sol
c65d491c9dcd287e87aa6a89e09e15f62ad35b59db10a693d92278295a5366e1  ./contracts/SolidlyLPStrat.sol
f3a3edf9a6b2e9e77bb5a1eaa792b2e2909e6eb9d59254bd7bd10d3a54913b32  ./contracts/PiGlobal.sol
97350f1e2177847efb8adb63d759237ba93c67a3a8c3a09526a6332b3ecdd212  ./contracts/StratAbs.sol
a1d8a103457818ecc2910a4598417284b58be917989524f5b9d681274d27bd48  ./contracts/Controller.sol
97c9c842a4b7e429c79bc9204ea82991a362780c2b98f0e7ffc6e3b53b05d911  ./contracts/SwapperWithCompensationAbs.sol
8ebe7b3a867eb972cb6889a1f5d3a452f1864f4bc46580095291d7a9bfbade68  ./contracts/DToken.sol
047689e5aeb92f2b2b299efa494a90f5da7efab43be68e4a226762e678edd53f  ./contracts/CollateralPool.sol
1e509b49057cb4a267337d29271e7a9728bdff1988f1efdc6a143e4669022468  ./contracts/DummyStrat.sol
4417fdff1145aa4258ca236b4aac94726341143c58b1cd8d36c763209426da51  ./contracts/Oracle.sol
f7641be4d9f187281ff697fce1eb5b9e759350ba123e664584ad62fa595baf25  ./contracts/PiAdmin.sol
33797c8cf763919d0c4d1d0de639b6f9496ecb022d04f81b4fcced165485cc7a  ./contracts/MetaCurveStrat.sol
86aadfa2438fc3d5ab6cfb0a44847f89bf8792bea8d53fedc8e4379a3e0cc5e2  ./contracts/MStableStrat.sol
026984096108a333beacecba4926a7fed2c3a294446d40564a8b2086c8d81145  ./contracts/DebtSettler.sol
5f88f5102e428c1ca1499e432f8745987cb962c587e8d837895c52dd22192555  ./contracts/LToken.sol
09ac394a6b9c1a950d9eac01e9f632bd0c0eabbb967772ec1e12ae1f98110f43  ./contracts/LiquidityPool.sol
56492fe6ea361f441d8a81b38e0b6f868dfdb12b3f4d73600cdbbbe6149c036d  ./contracts/PiOracleSolidly.sol
e99a446028e4815cc03f1e137d82666f4c1a7cf610b49adcb5d430060cd8162f  ./contracts/SwapperWithCompensationSolidly.sol
```

Coinspect also reviewed the new updated implementation of the `DebtSettler` contract, that will be used to reduce the borrower's debt according to the rewards harvested by a strategy. Auditors identified a new scenario that causes some massive repayments to fail because of a miscalculation in the credit amounts, **PIN-31**. In relation to the Controller, Coinspect identified that deposits below the minimum deposit amount (that was meant to work as an accumulating limit) are no longer allowed triggering a reversal, **PIN-36**.

A new set of contracts was provided that declare a pragma version of 0.8.17. The 2PI team provided tests for each strategy, however Coinspect **strongly suggests** adding more integration tests covering more scenarios between pools, strategies, debt settling, among others. These tests would have detected some of the issues found by Coinspect during this audit.

The contracts provided implement strategies that interact with other DeFi protocols such as Curve, Balancer, Uniswap, Jarvis among others. **The code of those third party contracts integrated by the strategies was not in scope for this audit**.

Strategies have a modular structure that rely on a base strategy abstract contract, `StratAbs,` that implements core functionality present on each strategy. Coinspect identified that several functions that should be not implemented, leveraging from the abstract feature, are implemented with no logic inside returning default initialization values. This behavior might lead to unexpected scenarios if a virtual, yet with empty implementation, function is not overridden in the child of `StratAbs,` **PIN-32**. Also, because of this, several strategies are not able to transfer fees or distribute harvested rewards leaving those tokens locked inside the strategy contract, **PIN-30**.

Regarding the rewards harvesting functionally, Coinspect observed the harvest function is public and not access restricted. **2PI should consider limiting the harvest function to a specific account** in order to diminish the attackers' ability to manipulate the pools and swap operations triggered by the harvest for profit.

In addition, there is an administrator role that is able to call specific access controlled functions that allow setting key parameters as well as handling strategy boosting. Coinspect identified that it is possible to cause an underflow when calculating minimum strategy share price triggered if the result of adding of the slippage ratio and the minimum virtual price is set over `1e18`, **PIN-34**. The migration process of strategies harvests rewards before changing the strategy address, however, Coinspect identified that for some strategies rewards might remain unclaimed during this process, **PIN-35**.

Each strategy performs a set of deposits and withdrawals on each underlying DeFi protocol with the goal to generate an additional yield on top of the borrowing and depositing rates that each 2PI pool provides. Coinspect identified that several pools could be drained by a read-only-reentrancy attack in the event of adding new compounding opportunities by enabling LP tokens as a collateral, **PIN-33**.

3. Summary of Findings

Id	Title	Total Risk	Fixed
PIN-30	Some strategies can't charge fees or transfer harvested rewards	High	✔
PIN-31	Denial of service when massively repaying debt	Medium	✔
PIN-32	Non implemented abstract functions break strategy functionality	Medium	✔
PIN-33	Liquidity pool can be drained if LP tokens are used as collateral	Medium	!
PIN-34	Potential underflow when calculating minWantAmount	Low	✔
PIN-35	JarvisStrat might not claim rewards when migrating	Low	✔
PIN-36	Controller prevents deposits below the threshold	Low	✔
PIN-37	Unfair charge of performance fees	Info	✔
PIN-38	Hardcoded addresses in contracts	Info	✔
PIN-39	Massive repay reverts due to USDT approve condition	Medium	✔

4. Detailed Findings

PIN-30 Some strategies can’t charge fees or transfer harvested rewards

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

Description

Tokens won't be transferred to the `Treasury` and `DebtSettler` for some strategies as key function’s implementations are empty, leaving those tokens locked inside the strategy contract.

Before performing any action in a strategy, the `_beforeMovement()` hook is called in order to transfer the performance fees to the Treasury and the remaining token surplus to the Debt Settler. This function internally calls `_withdrawFromPool()` and `_balanceOfPoolToWant()`. However, they have no logic for `Balancer` and `MetaCurve` strategies as their implementation is not overridden, not reverting and returning just zero respectively:

```
function _beforeMovement() internal virtual {
        uint _currentPoolBalance = _balanceOfPoolForMovement();
        uint _currentBalance = wantBalance();

        if (_currentPoolBalance > lastBalance) {
            uint _diff = _currentPoolBalance - lastBalance;
            uint _perfFee = _diff * performanceFee / RATIO_PRECISION;

            // Prevent to raise when perfFee is super small
            if (_perfFee > 0 && _balanceOfPoolToWant(_perfFee) > 0) {
                _withdrawFromPool(_perfFee);

                uint _feeInWant = wantBalance() - _currentBalance;

                if (_feeInWant > 0) {
                    want.safeTransfer(treasury, _feeInWant);
                    emit PerformanceFee(_feeInWant);
                }
            }

            // If debtSettler is set, we should sent all the generated balance to it
            // after charge protocol fees.
            if (address(debtSettler) != address(0)) {
                uint _extra = _diff - _perfFee;
```

```
            if (_extra > 0 && _balanceOfPoolToWant(_extra) > 0) {
                // withdraw all the extra generated tokens in pool since lastBalance
                _withdrawFromPool(_extra);

                uint _extraInWant = wantBalance() - _currentBalance;
                // send to debtSettler all extra tokens since lastMovement
                if (_extraInWant > 0) {
                    want.safeTransfer(debtSettler, _extraInWant);

                    emit DebtSettlerTransfer(_extraInWant);
                }
            }
        }
    }
}
```

Because `_balanceOfPoolToWant()` returns always zero, the call never enters any conditional branch where its value should be greater than zero.

As those functions are strictly implemented, the compiler will allow the deployment of the child as it considers that they already have logic without forcing the deployer to implement them:

```
function _withdrawFromPool(uint) internal virtual {
    // revert("Should be implemented");
}

function _balanceOfPoolToWant(uint) internal virtual view returns (uint) {
    // revert("Should be implemented");
}
```

Recommendation

Ensure that all functions are overridden on each child by not implementing any abstract function in `StratAbs`. Also, implement the missing functions.

Status

Fixed on commit `bd7e5dd5cb753fe520a257bce591d564aeeba0b8`.

The named functions are not implemented anymore, declaring only their signature in the abstract contract.

PIN-31 Denial of service when massively repaying debt

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

Description

A big borrow position is capable of stopping the massive repayment process by transferring more tokens than the minted debt.

The massive build and pay features of the `DebtSettler` were split on commit `3b5beef31d1d1eb7e11d1e74a017a3a9b5fdd998` preventing out of gas issues. This system works by tracking the last built and paid user's index of the enumerable borrowers mapping. However because those indexes are not controlled by the handler, an unexpected reordering of transactions when massive debt is being built will consequently break the credit calculation.

If a big borrower repays a part of his debt during a massive debt building, it will modify the total supply of debt and interest tokens, altering how credit is calculated for the next users. Also, if the `DebtSettler` has a limited amount of tokens (because of this miscalculation) it will run out of repayment assets reverting the whole current `pay()` loop.

The `build()` function has the following strategy to calculate the amount of credit among borrowers:

```
function build() external onlyHandler nonReentrant {
   ...
    uint _totalDebt = dToken.totalSupply() + iToken.totalSupply();
    if (_totalDebt == 0) return;

    // will always use first the "last balance snapshot"
    uint _amount = _lastCredit;
    // If there's a new loop we save the current balance
    if (_lastIndexBuilt == 0) _amount = _lastCredit = asset.balanceOf(address(this));

    if (_amount == 0) return;
    if (_amount > _totalDebt) _amount = _lastCredit =  _totalDebt;
   ...

    for (i; i < _length; i++) {
        address _borrower = _borrowers.at(i);
        uint _credit = _amount * _debt(_borrower) / _totalDebt;
   ...
  }
```

In addition, the `_debt()` function calculates the balance of interest and debt tokens for a specific borrower. Note that these functions will only account for already minted tokens and does not consider unminted interest tokens.

When calling `liquidityPool.repayFor()` inside `debtSettler.pay()`, the previously calculated credit for a user is passed:

```
function pay() external onlyHandler nonReentrant {
    ….
    for (i; i < _usersCredit.length(); i++) {
        // We should check for gasleft here, so we can repay the rest in the next tx if needed
        (address _borrower, uint _credit) = _usersCredit.at(i);

        if (_credit > 0 ) {
            if (dToken.balanceOf(_borrower) > 0) pool.repayFor(_borrower, _credit);
            _usersCredit.set(_borrower, 0);
        }
     …
    }
```

Because of this, every single `repayFor()` call burns less debt tokens than the effective amount transferred to the liquidity pool as it is decreased by the `_diff`.

```
function _repay(address _payer, address _account, uint _amount) internal {
   …
   if (_amount >= _totalDebt) {
    …
    } else {
        // In case of amount <= diff || amount <= (diff + iTokens)
        _interestToBePaid = _amount;

        if (_amount <= _diff) {
        …
        } else {
            _rest -= _diff;

            if (_rest <= _iTokens) {
            …
            } else {
                // Pay all the interests
                if (_iTokens > 0) iToken.burn(_account, _iTokens);

                _rest -= _iTokens;
                _interestToBePaid = _diff + _iTokens;

                // Pay partially the debt
                dToken.burn(_account, _rest);
            }
        }

        // Update last user interaction (or ending timestamp)
        uint _newTs = block.timestamp;
        if (_newTs > dueDate) _newTs = dueDate;
        _timestamps[_account] = uint40(_newTs);
    }

    // Take the payment
    asset.safeTransferFrom(_payer, address(this), _amount);
   …
  }
```

In the following section, the output of a test developed by Coinspect is provided. The test performed is a modification of the `“increases gas spent while building when there are more borrowers”` test provided by the 2Pi team. Coinspect added

a big borrower in the beginning and limited the `DebtSettler` asset funding to `10 * 10**18`.

Output

```
        CALL   LiquidityPool(0xdc64a140aa3e981100a9beca4e685f962f0cf6c9).repayFor(_account:   [CollateralPool],   _amount:
971815946213722)
         STATICCALL 2pi-D-t(0xb0279db6a2f1e01fbc8483fccef0be2bc6299cc3).balanceOf(account: [CollateralPool]) => (0.001)
         STATICCALL 2pi-D-t(0x3de2da43d4c1b137e385f36b400507c1a24401f8).balanceOf(account: [CollateralPool]) => (0.0)
         STATICCALL 2pi-D-t(0xb0279db6a2f1e01fbc8483fccef0be2bc6299cc3).balanceOf(account: [CollateralPool]) => (0.001)
         CALL 2pi-D-t(0xb0279db6a2f1e01fbc8483fccef0be2bc6299cc3).burn(_from: [CollateralPool], _amount: 971815933846903)
         EVENT
LiquidityPool(0xdc64a140aa3e981100a9beca4e685f962f0cf6c9).UnknownEvent(0x0000000000000000000000000000000000000000000000000
000373dc87862577, [0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef, [CollateralPool], 0x00])
         CALL   t(0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9).transferFrom(from:   [DebtSettler],   to:   [LiquidityPool],
amount: 971815946213722)


REVERT Error(reason: "ERC20: transfer amount exceeds balance")
```

Test

```
  it('REPAYMENT - debt building process DoS', async function () {
    const fixtures = await loadFixture(deploy)
    const {
      alice,
      bob,
      cPool,
      lPool,
      debtSettler,
      token,
      treasury,
    } = fixtures
    await setupCollateral(fixtures)
    // Add liquidity & Repayment
    await token.mint(lPool.address, 50e18 + '')
    await lPool.togglePause();

    // Bob incurs in a big debt before the build call
    // generate token balance
    await token.mint(bob.address, utils.parseEther("1000"))
    // approve and deposit
    await token.connect(bob).approve(cPool.address, utils.parseEther("100"))
    await expect(cPool.connect(bob)['deposit(uint256)'](utils.parseEther("100"))).to.emit(cPool, 'Deposit')
    await lPool.connect(bob).borrow(utils.parseEther("10"))

    let bobsDebt = await lPool['debt(address)'](bob.address)

    expect(bobsDebt).to.be.eq(utils.parseEther("10"))
    console.log(`0) Total Debt: ${(await lPool.totalDebt()).toString()}`);

    // Rest of borrowers
    let amountOfBorrowers = 300
    let borrowers = [];


    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)
      borrowers.push(curSigner);

      // 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')
      await lPool.connect(curSigner).borrow(1e15 + '')

      let signerDebt = await lPool['debt(address)'](curSigner.address)
```

```
      expect(signerDebt).to.be.eq(1e15)
    }


    // Since bob is the first in the enumerable mapping, his debt position is built in the first call
    await token.mint(treasury.address, 100e18 + '')
      await token.connect(treasury).transfer(debtSettler.address, ethers.utils.parseEther('10')) // Amt less than total
debt

    console.log(`1) Total Debt: ${(await lPool.totalDebt()).toString()}`);
    bobsDebt = await debtSettler._debt(bob.address);
    console.log(`Bobs debt: ${bobsDebt.toString()}`)

    let buildTx      = await debtSettler.connect(treasury).build({gasLimit: 10e6 })
    let buildReceipt = await buildTx.wait()

    expect(buildReceipt.gasUsed).to.be.greaterThan(9e6)

    console.log(`2) Total Debt: ${(await lPool.totalDebt()).toString()}`);

    // Bob now decides to repay his debt
    await token.connect(bob).approve(lPool.address, utils.parseEther("100"))
    await lPool.connect(bob).repay(utils.parseEther("0.01")) // Repays a part of his debt
    bobsDebt = await lPool['debt(address)'](bob.address)
    console.log(`Bobs debt: ${bobsDebt.toString()}`)


    buildTx      = await debtSettler.connect(treasury).build({gasLimit: 10e6 })
    buildReceipt = await buildTx.wait()


    expect(buildReceipt.gasUsed).to.be.greaterThan(7e6)

    console.log(`3) Total Debt: ${(await lPool.totalDebt()).toString()}`);
    buildTx = await debtSettler.connect(treasury).build({gasLimit: 10e6 })
    buildReceipt = await buildTx.wait()

    expect(buildReceipt.gasUsed).to.be.lessThan(5e6) // last only process 50 borrowers so should be left a few

    console.log(`Balance of Settler: ${(await token.balanceOf(debtSettler.address)).toString()}`);
    let buildReceipt2 = await (await debtSettler.pay({gasLimit: 10e6})).wait()
    expect(buildReceipt2.gasUsed).to.be.greaterThan(8e6)

    console.log(`Balance of Settler: ${(await token.balanceOf(debtSettler.address)).toString()}`);
    buildReceipt2 = await (await debtSettler.pay({gasLimit: 10e6})).wait()
    expect(buildReceipt2.gasUsed).to.be.lessThan(2e6)

    // this one will run without paying
    console.log(`Balance of Settler: ${(await token.balanceOf(debtSettler.address)).toString()}`);
    buildReceipt2 = await (await debtSettler.pay({gasLimit: 10e6})).wait()
    expect(buildReceipt2.gasUsed).to.be.lessThan(6e6)

    // Check if the other users had their debt repaid
    for (let index = 0; index < amountOfBorrowers; index++) {
      // get a signer
      let curSigner = borrowers[index];
      // add it to Hardhat Network
      curSigner = curSigner.connect(ethers.provider)
      let signerDebt = await lPool['debt(address)'](curSigner.address)
      expect(signerDebt).to.be.gt(0)
    }
    bobsDebt = await lPool['debt(address)'](bob.address)
    console.log(`Bobs debt: ${bobsDebt.toString()}`)
  })
```

Recommendation

Redesign the massive debt and payment mechanism using a time weighted system taking into account the mismatches between the debt and interest supply with the total debt (that considers unminted tokens).

Status

Fixed on commit f1ad8dee558d8333e077ec6191012ad7f246823c.

A new time weighted mechanism was implemented preventing the scenario shown before. This implementation also fixes the PIN-13 issue.

PIN-32 Non implemented abstract functions break strategy functionality

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

Description

Not overriding and implementing key strategy functions won't revert, leading to unexpected scenarios.

The `StratAbs` contract is a base abstract implementation for each strategy that provides the logic for key functions. However, its implementation is error prone as it does not take advantage of the property of abstract contracts which allows declaring functions as a part of its interface without implementing them:

```
function _deposit() internal virtual {
    // revert("Should be implemented");
}

function _withdraw(uint) internal virtual returns (uint) {
    // revert("Should be implemented");
}

function _withdrawAll() internal virtual returns (uint) {
    // revert("Should be implemented");
}

function _claimRewards() internal virtual {
    // revert("Should be implemented");
}

function _withdrawFromPool(uint) internal virtual {
    // revert("Should be implemented");
}

function _balanceOfPoolToWant(uint) internal virtual view returns (uint) {
    // revert("Should be implemented");
}

function balanceOfPool() public view virtual returns (uint) {
    // should be implemented
}

function balanceOfPoolInWant() public view virtual returns (uint) {
    // should be implemented
}
```

Because of this, future childs that do not implement those functions could be deployed without overriding any important function. The examples shown above return default initialization values or don't revert when called.

Recommendation

Only declare the functions as virtual without implementing them.

Status

Fixed on commit `bd7e5dd5cb753fe520a257bce591d564aeeba0b8`.

Same comment as PIN-30.

PIN-33 Liquidity pool can be drained if LP tokens are used as collateral

Total Risk	Impact	Location
Medium	High	`JarvisStrat.sol` `CollateralPool.sol`
Fixed !	Likelihood Low	

Description

Allowing LP tokens as collateral to provide compounding opportunities could be abused to drain other pools. The recently added `JarvisStrat` interacts with a Curve pool by adding and removing liquidity as well as depositing and withdrawing from Jarvis. If in the future any LP token is enabled as collateral, other liquidity pools could be drained because of a read-only reentrancy attack.

Curve tokens don't have a reentrancy guard for `get_virtual_price()`, `calc_token_amount()` and `calc_withdraw_one_coin()`. Also, the `remove_liquidity()` function performs the token transfer before updating the balance of the user. An attacker can request a flash loan, add liquidity and then remove it. The removal could trigger a fallback where the balances are still outdated in Curve inflating the collateral's price. This process enables a read only reentrancy.

Midas Capital was recently attacked because of this mistake. The attacker was able to steal $660k because Midas **enabled** `WMATIC-stMATIC Curve LP` token as collateral.

This issue was not further investigated because of time constraints.

Recommendation

Do not use Curve LP tokens as a collateral. Ensure that all strategies and pools are bulletproof to the behavior depicted on this issue.

Status

Acknowledged. The 2PI team said that it will consider this issue when handling LP tokens to prevent the impact mentioned before.

PIN-34 Potential underflow when calculating minWantAmount

Total Risk	Impact	Location

Low	Low	`JarvisStrat.sol`
		`MetaCurveStrat.sol`
		`StratAbs.sol`

Fixed ✔	Likelihood Low	

Description

An underflow could be triggered when calculating the `minWant` amount, causing a revert upon deposit and withdrawal.

The administrator can set both a `poolSlippageRatio` and `poolMinVirtualPrice`. Those variables should be smaller than `RATIO_PRECISION = 10000` when setting them separately. However, it is not checked that the addition of both parameters is less than the `RATIO_PRECISION` when setting them. Because of this, an underflow could be triggered:

```
    function _minWantToWantCrv(uint _amount) internal view returns (uint) {
       // Based on virtual_price (poolMinVirtualPrice) and poolSlippageRatio
       // the expected amount is represented with 18 decimals as crvWant token
       // so we have to add 12 decimals (on USDC and USDT for example) to the want balance.
       // E.g. 1e6 (1WANT) * 1e12 * 99.4 / 100.0 => 0.994e18 crvToken tokens
                return _amount * WANT_MISSING_PRECISION * (RATIO_PRECISION - poolSlippageRatio -
poolMinVirtualPrice) / RATIO_PRECISION;
    }

    function setPoolMinVirtualPrice(uint _ratio) public onlyAdmin {
        require(_ratio != poolMinVirtualPrice, "Same ratio");
        require(_ratio <= RATIO_PRECISION, "Can't be more than 100%");

        poolMinVirtualPrice = _ratio;
    }

    function setPoolSlippageRatio(uint _ratio) public onlyAdmin {
        require(_ratio != poolSlippageRatio, "Same ratio");
        require(_ratio <= RATIO_PRECISION, "Can't be more than 100%");

        poolSlippageRatio = _ratio;
    }
```

Recommendation

Check that the addition between `poolSlippageRatio` and `poolMinVirtualPrice` is smaller than `RATIO_PRECISION` on each setter.

Status

Fixed on commit `a1c58ebb118e7ea6086ccf6bec01f406a20ede6b`.

Both setters now check that no underflow could happen when adding the slippage with the minimum virtual price.

PIN-35	JarvisStrat might not claim rewards when migrating		
Total Risk **Low**	Impact Low	Location `JarvisStrat.sol`	
Fixed ✔	Likelihood Low		

Description

Rewards could remain unclaimed when migrating the strategy of a controller, without transferring them to the Debt Settler and Treasury.

The migration process of a strategy calls the `strategy.retireStrat()` which internally executes the reward harvesting process. This process, for the Jarvis strategy does not claim any reward if the strategy's end block has not passed yet:

```
function _claimRewards() internal override {
    uint pending = IJarvisPool(JARVIS_POOL).pendingRwd(JARVIS_POOL_ID, address(this));

    if (pending > 0) {
        IJarvisPool(JARVIS_POOL).deposit(JARVIS_POOL_ID, 0);
    }

    // If the endBlock is reached we burn all the AG_DENARIUS tokens to get rewards
    if (IJarvisPool(JARVIS_POOL).endBlock() <= block.number) {
        uint bal = IERC20(AG_DENARIUS).balanceOf(address(this));

        if (bal > 0) {
            IERC20(AG_DENARIUS).safeApprove(REWARDS_STORAGE, bal);
            IJarvisRewards(REWARDS_STORAGE).claim(bal);
        }
    }
}
```

As a result of this, no rewards are claimed and the migration process succeeds.

Recommendation

Ensure that no rewards will remain unclaimed during the migration process of all strategies.

Status

Won't fix.

The 2PI team stated that this function will be used only for emergencies, then it would be not desirable to prevent the migration because of the rewards.

PIN-36 Controller prevents deposits below the threshold

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

Description

The strategy deposit threshold triggers a revert if a deposit below that amount is made, forcing users to deposit at least that amount.

The accumulation threshold was added to the Controller aiming to fix `PIN-22: Strategy hooks in controller will reduce profitability of operations.` The nature of that fix was meant to work as an accumulator, allowing deposits below the threshold and calling the strategy only when that deposit threshold is exceeded. However, the implementation reverts instead of controlling the flow of the calls:

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

    uint _amount = assetBalance();

    if (_amount < minStrategyDepositAmount) revert MinDepositAmountNotReached();

    asset.safeTransfer(address(strategy), _amount);

    strategy.deposit();
}
```

Instead of reverting, it should create a conditional branch that is executed only when the `minStrategyDepositAmount` is exceeded in order to work as an accumulator. Also, this implementation won't stop the Controller from calling the strategy each time as no accumulation is being made.

Recommendation

Execute the external calls inside the conditional branch if the condition is met instead of reverting.

Status

Fixed on commit 476a197e36d2f37312129478e9b1e4b8ee309cac.

Instead of reverting, a conditional check was included transferring the Controller's balance only when the threshold is passed.

PIN-37 Unfair charge of performance fees

Total Risk	Impact	Location
Info	-	StratAbs.sol
Fixed ✔	Likelihood -	

Description

Performance fee rate could be modified suddenly before harvesting, altering the amount of fees charged.

Each time a `harvest()` call is made, fees are calculated and transferred to the treasury This function is public and is expected to be called frequently. However, if the frequency is reduced (e.g. during a congested network scenario), the amount of claimed rewards could be higher making this process even more profitable for the admin.

```
function _chargeFees(uint _harvested) internal {
   uint _fee = (_harvested * performanceFee) / RATIO_PRECISION;

   // Pay to treasury a percentage of the total reward claimed
   if (_fee > 0) {
       want.safeTransfer(treasury, _fee);
       emit PerformanceFee(_fee);
   }

   if (address(debtSettler) != address(0)) {
       uint _extra = _harvested - _fee;

       // send to debtSettler all extra tokens since lastMovement
       if (_extra > 0) {
           want.safeTransfer(debtSettler, _extra);
           emit DebtSettlerTransfer(_extra);
       }
   }
}
```

A sudden call to `setPerformanceFee()` in this scenario, will set a new fee and it will consider the unharvested amount when calculating new fees making the calculation retroactively.

Recommendation

Call `harvest()` before modifying the performance fee.

Status

Fixed on commit `60b0f19055be83bf2f68af58815814abab37b48d`.

Now `harvest()` is called before changing the performance fee.

PIN-38	Hardcoded addresses in contracts		
Total Risk **Info**	Impact -	Location -	
Fixed ✔	Likelihood -		

Description

Deploying a contract that has hardcoded addresses in a different chain as intended will break the functioning of the strategy.

Some contracts have hardcoded addresses and several chains are supported according to the `hardhat.config.js` file. In the event of deploying a contract in a mistaken chain, it won't work as intended leading to unexpected scenarios.

Recommendation

Include NatSpec mentioning the chain where a contract is meant to be deployed along with any other relevant information.

Status

Fixed on commit `2887a7a2bedbdad95ae8fefcb67a5bc59aa2a6ad`.

PIN-39 Massive repay reverts due to USDT approve condition

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

Description

Unhandled approval condition of some tokens lead to failed `DebtSettler.pay()` calls, disrupting the massive debt repayment feature.

Some tokens like USDT, enforce that the allowance must be set to zero before setting it to a new non-zero value, as a mitigation for the `"ERC20 approve race condition"` vulnerability. If a smart contract calls the `approve()` function without handling this condition the transaction may be reverted, affecting the underlying logic.

DebtSettler.pay()

```
    function pay() external onlyHandler nonReentrant {
        // Ensure always pay after build is finished
        if (_lastIndexBuilt > 0) revert StillBuilding();

        asset.approve(address(pool), _lastCredit);
        // continues here...
    }
```

USDT.approve()

```
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Recommendation

Check and reset the allowance to zero before setting a new non-zero value.

Status

Fixed on commit `5141af2e1a4916fee24d63ce9572f780509b4e14`.

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.