Redacted Protocol Pirex-Btrfly Audit

Introduction & Scope

This audit looks into Pull Request 1 as seen here. This includes the following contracts:

- `PirexBtrfly.sol`
- `PirexBtrflyBase.sol`
- `PirexFees.sol`
- `PxBtrfly.sol`
- `UnionPirexVault.sol`
- `UnionPirexStrategy.sol`
- `UnionPirexStaking.sol`

This audit was conducted by kebabsec members sai, FlameHorizon and okkothejawa.

Note: This report does not provide any guarantee or warranty of security for the project.

The following report may contain 0-day vulnerabilities in existing live projects such as PirexCvx and Union vaults, thus report shouldn't be released or circulated before it is ensured that these vulnerabilities are patched. Kebabsec has delegated the responsibility of the disclosure of these vulnerabilities to their respective teams to `PirexBtrfly` team.

Executive Summary

Table of contents

Findings:

1. [HIGH] Targeted denial of service of rewards redemption against a particular user

- In the ordinary flow of the system, a call to `redeemSnapshotRewards` for a particular user's reward of a certain epoch is preceded by a call to `claimRewards` for that epoch. As `claimRewards` actually gets the rewards of that epoch from Redacted's `RewardDistributor` and records it as the rewards metadata and use this metadata in the distribution, if a call to `redeemSnapshotRewards` is made before the call to `claimRewards`, that user won't receive the rewards they are entitled to.
- Following line of code ensures that a user cannot redeem the same reward token more than once in a given epoch, enabling this issue.

```
if ((redeemed & indexRedeemed) != 0) revert AlreadyRedeemed();
```

- If `redeemSnapshotRewards` had proper access control, this would only result in possible self-DOS. But as anyone can redeem anyone's rewards through `redeemSnapshotRewards`, an attacker can frontrun the call to the `claimRewards` and call `redeemSnapshotRewards` by passing in the victim(s) addresses as the `account` parameter. Due to the line of code above, this user wouldn't be able to redeem the reward token(s) they are entitled again, even if the rewards metadata was updated after the attacker's griefing attack.
- **Recommendation**: Lock `redeemSnapshotRewards` so that a caller can only redeem their own rewards, which would reduce this issue to only potential self-DOS. To mitigate self-DOS too, consider ensuring a call to `claimRewards` is always made before a redeem call.

2. [HIGH] System-wide denial of service of rewards redemption

- As can be seen in the issue above, the function `claimRewards` is responsible from pulling the rewards tokens from the Redacted protocol. This is done by an external call to `RewardDistributor` of Redacted in the throughout the execution of `claimRewards`:

```
            rewardDistributor.claim(params);
```

- The problem is, this `params` parameter is partially passed as a parameter to `claimRewards`, and partially set in the function, which would be known to an attacker beforehand. We are not certain if the passed in `merkleProof` can be known to an attacker beforehand, but even if this is not possible, the attacker can simply front-run the authentic call to `claimRewards`, construct their own `params` from it, and call the `rewardDistributor.claim()` on their own as the `claim` of `rewardDistributor` has no access control and anyone can trigger any account's reward distribution. As a seperate call to `RewardDistributor` would result in the rewards metadata not getting updated properly (as the pulled rewards wouldn't get accounted), rewards would be essentially locked in the Pirex system, and the system wouldn't be able to properly distribute these rewards to the users, resulting in overall denial of service of the rewards redemption process of Pirex.

- **Recommendation**: Consider restructuring `claimRewards` so that external calls like in the above scenario do not result in DOS.

3. [INFO] Lack of access control in `distributeFees` can lead to event pollution

- `distributeFees` of `PirexFees` has no access control, meaning that anyone can pull tokens from an address that has approved `PirexFees` and send the funds to treasury. We think this is not a security risk, as `PirexFees` is only approved in `PirexBtrfly` and the approval is consumed in its entirety afterwards in each occurrence. Yet, the emit that `distributeFees` can be polluted as anyone can emit the event as if the Pirex system is emitting fees in a fake token, which can be used to give credibility to phishing campaigns utilizing Etherscan and its variations .
- **Recommendation**: Consider adding `onlyVault` access control modifier to `distributeFees` .

4. [LOW] Unfair rounds calculation in `_initiateRedemption`

- `_initiateRedemption` grants an user 1 round of futures rewards if their unlock time is at a mid-epoch (so 1 week from the start of an epoch) and they have a wait time of 7 days to 14 days, as the reward rounds calculation is rounded down.

```
// Determine how many futures notes rounds to mint
      uint256 rounds = waitTime / EPOCH_DURATION;

      // Check if the lock was in the first week/half of an epoch
      // Handle case where remaining time is between 1 and 2 weeks
      if (
          rounds == 0 &&
          unlockTime % EPOCH_DURATION != 0 &&
          waitTime > (EPOCH_DURATION / 2)
      ) {
          // Rounds is 0 if waitTime is between 1 and 2 weeks
          // Increment by 1 since user should receive 1 round of rewards
          unchecked {
              ++rounds;
          }
      }
```

- Yet, if they wait for 27 days and 23 hours, they also get 1 round of rewards due to the same rounding down effect, which means an user who waited for 27 days can get the same amount of futures rewards with another one who waited for just 7 days and a minute, even though the difference of their wait times are more than an epoch (1 week).
- **Recommendation**: Consider rounding up for any wait time, or communicating this behavior through comments and or the docs.
- Pirex team told us that this is intended, yet we believe this issue might be useful for an user reading this, thus the issue is still kept in the report.

5. [INFO] `_mintFutures` defaults to `rpxBtrfly`

- `_mintFutures` , likely in the pursuit of gas optimization, defaults to minting `rpxBtrfly` , as it only checks if the provided `f` parameter is equal to `Futures.Vote` , even though `Futures.Reward` also exists in the `Futures` struct.

```
        ERC1155PresetMinterSupply token = f == Futures.Vote
            ? vpxBtrfly
            : rpxBtrfly;
```

- This means someone that calling `stake()` without filling `f` parameter would receive `rpxBtrfly` even though they didn't explicitly preferred it so.
- **Recommendation**: Consider requiring `f` parameter being necessarily either `Futures.Vote` or `Futures.Reward` .

6. [HIGH] Union staking funds can be stolen

- `UnionPirexVault` is an ERC4626 implementation, and thus it utilizes the `totalAssets()` view function to determine asset-to-share and share-to-asset conversions. As assets, or the `pxBtrfly` in this context, is stored in the `UnionPirexStaking` expected flow, `totalAssets()` gets the `_totalSupply` of the staking contract, which should be equal to the total `pxBtrfly` balance in the Union system that came through deposits, and gets `rewards` which stands for the rewards that has been accumulated for the overall Union system. Yet `totalAssets()` makes a wrong assumption, which can be seen in the following comment:

```
      // Vault assets + rewards should always be stored in strategy until withdrawal-time
```

- This assumption is incorrect as the `getReward` of `UnionPirexStaking` is permissionless, which transfers the accumulated `pxBtrfly` rewards to `UnionPirexVault` so it can be staked again and auto-compounded through a `harvest()` call. The problem is that, due to the permissionless nature of `getReward` , anyone can call the `getReward` function, and `pxBtrfly` would be sent to `UnionPirexVault` without getting deposited again, and `rewards` would return 0, breaking the invariant highlighted in the comment above. A call to `harvest()` afterwards would more or less restore the old value.
- The behavior explained above would give an attacker an ability to alter the return value of `totalAssets()` in an single transaction:

```
//Friction sources like fees and withdrawal penalty is ignored in the scenario below.

totalAssets -> 50
* Attacker calls getReward() *
totalAssets -> 25
* Attacker calls harvest() *
totalAssets -> 50
```

- Consider a more advanced attack scenario, with the same assumptions being made:

```
1)
_totalSupply (staking) -> 50
supply (vault) -> 50
rewards -> 50
totalAssets -> _totalSupply + rewards -> 100
2)
* Attacker calls getReward() *
_totalSupply (staking) -> 50
supply (vault) -> 50
rewards -> 0
totalAssets -> _totalSupply + rewards -> 50
3)
* Attacker makes a deposit of 50 assets, attack cost is 50 assets *
attacker's shares -> ((deposited asset amount) * supply) / totalAssets -> 50 * 50 / 50 -> 50

supply (vault) -> 100
_totalSupply (staking) -> 100
totalAssets -> 100
4)
* Attacker calls harvest(), the rewards are staked again *
_totalSupply(staking) -> 150
supply(vault) -> 100 (as no new shares are being minted in the ERC4626 vault, vault supply is not affected)
totalAssets -> _totalSupply + rewards -> 150 + 0 -> 150
5)
* Attacker calls redeem() with the 50 shares they got *
redeemed amount of assets -> (50 * totalAssets) / supply (vault) -> (50 * 150) / 100 -> 75

Attacker's cost was 50 assets, now they redeemed 75, 25 assets are attacker's profit
```

- Keep in mind that the practical severity of this issue is dependent on many factors, such as the frequency of the calls to `harvest()`.
- We think this issue can only result in the theft of yield (accumulated rewards) and not of an user's primary deposit, yet there might be ways to utilize this issue for the theft of the primary deposits that we couldn't discover.
- **Recommendation**: Make `getReward()` permissioned with the `onlyVault` modifier so that the invariant `Vault assets + rewards should always be stored in strategy until withdrawal-time` and the calls to `getReward()` necessiates the rest of the `harvest()` action.

7. [INFO] `setContract` defaults to `UnionPirexVault`

- Much like the issue 5 the function `setContract` defaults to setting `UnionPirexVault` address in when an explicit `c` parameter is not given, even though the `Contract` struct has `UnionPirexVault` in it. If the owner calls `setContract` without giving a proper `c` they might overwrite `UnionPirexVault` address on accident.
- **Recommendation:** Properly check if `c == Contract.UnionPirexVault` in an additional if branch, and and revert if no contract categories match with the `c` parameter.

8. [LOW] Unfair `spxBTRFLY` minting might disincentivize users from staking for longer periods of time

- In line 695 of `PirexBtrfly.sol` the amount of time staked is calculated for the minting of `spxBTRFLY` tokens, but with this calculation, even a user that staked any amount of time, as long as it was done in the current epoch and passes the end of that same epoch. This could be problematic as it could be unfair for some users that stake for longer get the same amount of spxBtrfly minted, or just the fact that the protocol will not get the benefit of staking from a population of users that will just understand this behaviour and stake right before an epoch ends, wait for the next block and reap the rewards without needing to actually lock any of their funds for any substancial amount of time.
- **Recommendation:** In this case, perhaps is best to have a smoother time to rewards curve, for example, the closer a user stakes at the end of an epoch, the least benefit they would get, inversely, the protocol could exponentially raise rewards for users that stake at the first hours of an epoch in order to incentivize early and longer staking.