REN∆SCENCE

Institutional Pirex ETH Audit Report

Version 1.0

Audited by:

MiloTruck

bytes032

April 10, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol’s core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol’s functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Dinero

Dinero is an experimental protocol which capitalizes on the premium blockspace market by introducing:

1. An ETH liquid staking token ("LST") which benefits from staking yield and the Dinero protocol
2. A decentralized stablecoin (DINERO) as a medium of exchange on Ethereum
3. A public and permissionless RPC for users

2.2 Overview

Project	Institutional Pirex ETH
Repository	dinero-pirex-eth
Commit Hash	53c0eef2f99b...
Date	1 April 2024 - 9 April 2024

2.3 Issues Found

Severity	**Count**
High Risk	1
Medium Risk	4
Low Risk	1
Informational	4
Total Issues	**10**

3 Findings Summary

ID	Description	Status
H-1	`executeInitiateRedemption()` incorrectly calls `initiateRedemption()` with `postFeeAmount`	Open
M-1	`maxBufferSize` isn't re-calculated when `maxBufferSizePct` changes in `setMaxBufferSizePct()`	Open
M-2	Partial redemptions should not be allowed when there are no staked validators left	Open
M-3	`updateBuffer()` burns pxETH instead of institutional pxETH	Open
M-4	`RewardRecipientGateway.slashValidator()` prevents legacy reward recipient from using the buffer with non-zero `_amount`	Open
L-1	`_depositSize` check in constructor of `InstitutionalPirexEthValidators` is incorrect	Open
I-1	Redundant check in `executeSetMaxProcessedValidatorCount()`	Open
I-2	Withdrawal credentials for non-institutional validators will be mixed	Open
I-3	Logic in `AutoPxEth.previewWithdraw()` can be simplified	Open
I-4	`redirectEnabled` should always be `false` for institutional `AutoPxEth`	Open

4 Findings

High Risk

[H-1] `executeInitiateRedemption()` incorrectly calls `initiateRedemption()` with `postFeeAmount`

Context:

- InstitutionalPirexEthWithdrawLogic.sol#L371-L381
- InstitutionalPirexEthWithdrawLogic.sol#L502-L506

Description: In `InstitutionalPirexEthWithdrawLogic.sol`, `executeInitiateRedemption()` calls `initiateRedemption()` with `postFeeAmount`:

```
initiateRedemption(
    // ...
    initiateRedemptionParams.depositSize,
    postFeeAmount,
    initiateRedemptionParams.receiver,
    initiateRedemptionParams.shouldTriggerValidatorExit
);
```

However, `_pxEthAmount` should be passed to `initiateRedemption()` instead of `postFeeAmount`.

`postFeeAmount` is the amount of institutional pxETH sent by the caller after fees have been subtracted, which has a 1:1 value with apxETH (note that the comment below is wrong):

```
institutionalPxEth.transferFrom(msg.sender, address(this), assets);

// Get the pxETH amounts for the receiver and the protocol (fees)
(postFeeAmount, feeAmount) = InstitutionalPirexEthGenericLogic
    .computeAssetAmounts(fees, feeType, assets);
```

However, `initiateRedemption()` expects the amount of pxETH to be redeemed, which would be `_pxEthAmount`.

This causes `initiateRedemption()` to mint less upxETH to users upon withdrawal, causing a loss of funds.

Recommendation: Pass `_pxEthAmount` instead of `postFeeAmount`:

```
  initiateRedemption(
      // ...
      initiateRedemptionParams.depositSize,
-     postFeeAmount,
+     _pxEthAmount,
      initiateRedemptionParams.receiver,
      initiateRedemptionParams.shouldTriggerValidatorExit
  );
```

Medium Risk

[M-1] `maxBufferSize` isnt re-calculated when `maxBufferSizePct` changes in `setMaxBufferSizePct()`

Context:

- InstitutionalPirexEthValidators.sol#L442-L452
- InstitutionalPirexEthDepositLogic.sol#L319-L324

Description: When `maxBufferSizePct` is modified by calling `setMaxBufferSizePct()`, the function does not re-calculate `maxBufferSize` according to the new value of `maxBufferSizePct`. Therefore, if `maxBufferSizePct` is decreased, `maxBufferSize` will be temporarily higher than what it should be.

This only becomes a problem when `slashValidator()` is called on a `Staking` validator immediately after `setMaxBufferSizePct()`.

`slashValidator()` calls `InstitutionalPirexEthDepositLogic.addPendingDeposit()` directly without re-calculating `maxBufferSize` beforehand:

```
InstitutionalPirexEthDepositLogic.addPendingDeposit(
    // ...
);
```

Since `maxBufferSize` is still inflated, `addPendingDeposit()` will then fill up the buffer according to the outdated value of `maxBufferSize`:

```
uint256 _remainingBufferSpace = (
    pirexEthValidatorVars.maxBufferSize > pirexEthValidatorVars.buffer
        ? pirexEthValidatorVars.maxBufferSize -
            pirexEthValidatorVars.buffer
        : 0
);
```

For example:

- Assume the following:
  - `buffer = 32 ETH`
  - `maxBufferSize = 64 ETH`
  - `maxBufferSizePct` is 30%
- Governance calls `setMaxBufferSizePct()` to decrease `maxBufferSizePct` to 15%:
  - If `maxBufferSize` was re-calculated, it would be 32 ETH.
- `slashValidator()` is called to slash a validator with `ValidatorStatus.Staking` status. In `addPendingDeposit()`:
  - `_remainingBufferSpace = 32 ETH`, therefore the ETH from the slashed validator is added to the buffer.
- However, if `maxBufferSize` was updated to 32 ETH, the ETH from the slashed validator would have been added to `pendingDeposit` and used to stake a new validator.

As seen from the example above, due to the outdated value of `maxBufferSize`, ETH was wrongly added to the buffer instead of being used to stake a new validator. This will cause the protocol to generate less yield.

Recommendation: Consider updating `maxBufferSize` in `executeSetMaxBufferSizePct()`:

```diff
  function executeSetMaxBufferSizePct(
      DataTypes.PirexEthValidatorVars storage pirexEthValidatorVars,
+     DataTypes.PirexEthValidatorContracts storage pirexEthValidatorContracts,
      uint256 pct
  ) external {
      InstitutionalPirexEthValidationLogic.validateSetMaxBufferSizePct(pct);
      emit SetMaxBufferSizePct(pct);
      pirexEthValidatorVars.maxBufferSizePct = pct;
+     pirexEthValidatorVars.maxBufferSize =
+         pirexEthValidatorContracts.pxEth.totalSupply() * pct /
Constants.DENOMINATOR;
  }
```

[M-2] Partial redemptions should not be allowed when there are no staked validators left

Context: InstitutionalPirexEthWithdrawLogic.sol#L469-L476

Description: In `InstitutionalPirexEthWithdrawLogic.sol`, `initiateRedemption()` adds `pxEthAmount` to `pendingWithdrawal` and proceeds to process pending withdrawals in multiples of `DEPOSIT_SIZE`.

If `pxEthAmount` is still non-zero afterwards, the function will initiate a partial redemption by minting IUpxEth with the `batchId` of a future validator:

```
if (pxEthAmount > 0) {
    pirexEthValidatorContracts.iupxEth.mint(
        receiver,
        pirexEthValidatorVars.batchId,
        pxEthAmount,
        ""
    );
}
```

However, allowing partial redemptions for validators that haven't been unstaked might be problematic in an exit scenario where everyone is trying to redeem their institutional pxETH (e.g. migration to a new set of contracts).

This is because there might be a case where IUpxEth is minted to a user for a future `batchId`, but there is no more staked validators to exit. As such, the IUpxEth can never be redeemed for ETH.

For example:

- Assume everyone is trying to exit `InstitutionalPirexEth`:
    - There is 1 staked validator left (32 ETH) and 31 ETH in the buffer.
    - Alice holds 31 ETH worth of IPxEth, Bob holds 32 ETH worth of IPxEth.
    - `pendingWithdrawal = 0`

- To avoid incurring the instant redemption fee, Alice calls `initiateRedemption()` with:
  - `_assets` as all her IPxEth
  - `_shouldTriggerValidatorExit = false`, since her 31 ETH is insufficient to trigger a validator exit
- Bob front-runs her transaction and calls `initiateRedemption()` to withdraw all his institutional pxETH:
  - This triggers the exit of the last validator, and his IUpxEth has the `batchId` of the last validator
- When Alice's transaction is processed:
  - `pendingWithdrawal + postFeeAmount = 31 ETH`, so `_requiredValidators = 0` and this check passes
  - Her withdrawal is a partial redemption, so the upxEth minted to her has the `batchId` of the next validator
- However, since there are no more validators to exit, her upxETH can never be redeemed.

Recommendation: In `InstitutionalPirexEthWithdrawLogic.initiateRedemption()`, consider reverting if there aren't any staked validators left:

```
  if (pxEthAmount > 0) {
+     if (stakingValidators.count() == 0) revert Errors.NoValidatorsLeft();
      pirexEthValidatorContracts.iupxEth.mint(
          receiver,
          pirexEthValidatorVars.batchId,
          pxEthAmount,
          ""
      );
  }
```

This prevents users from initiating partial redemptions for future validators when there are no staked validators remaining.

[M-3] `updateBuffer()` burns pxETH instead of institutional pxETH

Context: InstitutionalPirexEthConfigurationLogic.sol#L424-L435

Description: In `InstitutionalPirexEthConfigurationLogic.sol`, `batchBurnPxEth()`, which is called by `updateBuffer()`, directly burns pxETH from all addresses specified in the `burnerAccounts` array:

```
for (uint256 _i; _i < _len; ) {
    if (!burnerAccounts[$burnerAccounts[_i].account])
        revert Errors.AccountNotApproved();

    _sum += $burnerAccounts[_i].amount;

    burnPxEth(
        // ...
    );
```

However, since this is the institutional version of the protocol, it's not possible for any address to gain pxETH - depositing ETH through `InstitutionalPirexEth.deposit()` mints institutional pxETH to the caller, which cannot be unwrapped into apxETH unless you have the `BURNER_ROLE`.

The only address that holds pxETH would be the `AutoPxEth` contract, which makes it impossible for governance to use burner accounts to compensate ETH.

Recommendation: The function should burn institutional pxETH from burner accounts instead, by:

- Transferring institutional pxETH from the burner account to this contract.
- Unwrapping institutional pxETH into apxETH.
- Redeeming apxETH for pxETH and burning the received amount.

[M-4] `RewardRecipientGateway.slashValidator()` prevents legacy reward recipient from using the buffer with non-zero `_amount`

Context:

- RewardRecipientGateway.sol#L146-L148
- RewardRecipient.sol#L130-L137

Description: `RewardRecipientGateway.slashValidator()` is meant to be called from two addresses - directly by the `KEEPER_ROLE`, or from the legacy `RewardRecipient` contract's `slashValidator()` function.

When calling `RewardRecipientGateway.slashValidator()` with `_useBuffer = true`, `msg.value` has to be zero:

```
if (_useBuffer && msg.value > 0) {
    revert Errors.NoETHAllowed();
}
```

However, this check makes it impossible to call `RewardRecipient.slashValidator()` with `_useBuffer = true` and a non-zero `_amount`.

When calling this function, `RewardRecipient.slashValidator()` forwards `msg.value + _amount`:

```
pirexEth.slashValidator{value: _amount + msg.value}(
    // ...
);
```

Therefore, when this check is reached, `msg.value` will not be `0` if `_amount` is non-zero, causing this check to fail.

As such, the keeper will not be able to dissolve a validator using ETH unstaked from the validator and the buffer, which will be required in scenarios where validators are partially penalized.

Recommendation: Consider enforcing this check only when `slashValidator()` is not called through the legacy `RewardRecipient` contract:

```
- if (_useBuffer && msg.value > 0) {
+ if (msg.sender != legacyRewardRecipient && _useBuffer && msg.value > 0) {
      revert Errors.NoETHAllowed();
  }
```

Low Risk

[L-1] `_depositSize` check in constructor of `InstitutionalPirexEthValidators` is incorrect

Context: InstitutionalPirexEthValidators.sol#L174-L175

Description: The constructor of `InstitutionalPirexEthValidators` contains the following check:

```
if (_depositSize < 1 ether && _depositSize % 1 gwei != 0)
    revert Errors.ZeroMultiplier();
```

This check is meant to enforce that `_depositSize` is not less than 1 ETH and is a multiple of 1 gwei. However, it incorrectly uses `&&` instead of `||`, as such, one condition could be false and the check would still pass. For example, `_depositSize = 0` would pass this check.

Recommendation: Modify the check to use `||`:

```
- if (_depositSize < 1 ether && _depositSize % 1 gwei != 0)
+ if (_depositSize < 1 ether || _depositSize % 1 gwei != 0)
      revert Errors.ZeroMultiplier();
```

Informational

[I-1] Redundant check in `executeSetMaxProcessedValidatorCount()`

Context: InstitutionalPirexEthConfigurationLogic.sol#L217-L221

Description: `InstitutionalPirexEthConfigurationLogic.executeSetMaxProcessedValidatorCount()` checks that `count` is non-zero:

```
if (count == 0) {
    revert Errors.InvalidMaxProcessedCount();
}
InstitutionalPirexEthValidationLogic
    .validateSetMaxProcessedValidatorCount(count);
```

However, this check is redundant as `InstitutionalPirexEthValidationLogic.validateSetMaxProcessedValidatorCount()` performs the exact same check.

Recommendation: Remove the `count == 0` check:

```
- if (count == 0) {
-     revert Errors.InvalidMaxProcessedCount();
- }
  InstitutionalPirexEthValidationLogic
      .validateSetMaxProcessedValidatorCount(count);
```

[I-2] Withdrawal credentials for non-institutional validators will be mixed

Context: PirexEthValidators.sol#L515-L521

Description: In the new design of `RewardRecipientGateway`, the contract calls functions in the currently deployed `PirexEth` contract. This means that `PirexEthValidators.rewardRecipient` will have to be changed to point to the `RewardRecipientGateway` contract.

However, this would change the `withdrawalCredentials` in the `PirexEthValidators` contract as well:

```
} else if (_contract == DataTypes.Contract.RewardRecipient) {
    rewardRecipient = contractAddress;
    withdrawalCredentials = abi.encodePacked(
        bytes1(0x01),
        bytes11(0x0),
        contractAddress
    );
```

As such, new validators initialized using `PirexEthValidators.addInitializedValidators()` will have withdrawal credentials pointing to this contract, instead of the legacy `RewardRecipient` contract.

This is somewhat problematic, since the existing validators will continue to send rewards to the legacy `RewardRecipient`, while the new validators from the non-institutional protocol will send rewards to `RewardRecipientGateway`.

Therefore, the keeper might have to call functions in `RewardRecipientGateway` directly to handle non-institutional validators.

[I-3] Logic in `AutoPxEth.previewWithdraw()` can be simplified

Context: AutoPxEth.sol#L415-L419

Description: The following calculation in `AutoPxEth.previewWithdraw()` can be replaced `super.previewWithdraw(assets)` since `ERC4626.previewWithdraw()` performs the same logic:

```
// Calculate shares based on the specified assets' proportion of the pool
uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is non-zero.
uint256 shares = supply == 0
    ? assets
    : assets.mulDivUp(supply, totalAssets());
```

Recommendation:

```diff
  // Calculate shares based on the specified assets' proportion of the pool
- uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is non-zero.
- uint256 shares = supply == 0
-     ? assets
-     : assets.mulDivUp(supply, totalAssets());
+ super.previewWithdraw(assets);
```

[I-4] `redirectEnabled` should always be `false` for institutional `AutoPxEth`

Context:

- AutoPxEth.sol#L497-L499
- InstitutionalPirexEthWithdrawLogic.sol#L508-L509

Description: When `redirectEnabled` is set to true, transferring apxETH to the `InstitutionalPirexEth` contract using `AutoPxEth.transferFrom()` will call `InstitutionalPirexEth.initiateRedemption()`:

```
if (redirectEnabled && to == address(pirexEth)) {
    pirexEth.initiateRedemption(amount, from, false);
}
```

However, for the institutional version of the protocol, `redirectEnabled` should always be set to `false`.

If it is set to `true`, in `InstitutionalPirexEthWithdrawLogic.sol`, both `instantRedeemWithIPxEth()` and `executeInitiateRedemption()` would not be callable.

This is because both functions eventually call `institutionalPxEth.unwrap()`, which calls `AutoPxEth.transferFrom()`. As such, this block would be triggered, calling `initiateRedemption()` again and reverting due to the reentrancy lock.