Dinero (Branded LST) Audit Report

Version 1.0

Audited by:

MiloTruck

bytes032

June 24, 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	Dinero (Branded LST)
Repository	dinero-pirex-eth
Commit Hash	3f926198a1f8...
Mitigation Hash	be1b48358d27...
Date	7 June 2024 - 12 June 2024

2.3 Issues Found

Severity	**Count**
High Risk	7
Medium Risk	6
Low Risk	1
Informational	0
Total Issues	**14**

3 Findings Summary

ID	Description	Status
H-1	`DineroERC20RebaseUpgradeable._transferShares()` duplicates shares on self-transfer	Resolved
H-2	`LiquidStakingToken.mint()` doesn't add deposit fee shares to `RateLimiter.unsyncedLimit`	Resolved
H-3	`LiquidStakingToken.withdraw()` subtracts the amount of assets to withdraw from `_totalAssets()` twice	Resolved
H-4	`LiquidStakingTokenLockbox._calculateWithdrawalAmount()` excluding `pendingDeposit` causes loss of funds on withdrawal	Resolved
H-5	Incorrect check in `RateLimiter.updateRateLimit()` causes `unsyncedLimit` balance to be frozen	Resolved
H-6	Total assets in `LiquidStakingToken` is wrongly reduced twice for a withdrawal	Resolved
H-7	`totalStaked` is not increased for finalized L2 deposits	Resolved
M-1	`L1SyncPoolETH._finalizeDeposit()` does not account for the `PirexEth` deposit fee	Resolved
M-2	`LiquidStakingToken` and `LiquidStakingTokenLockbox` are not pausable	Resolved
M-3	Depositors unfairly accrue yield from the `assetsPerShare` difference when minting modeETH	Resolved
M-4	Rounding for share calculation in `LiquidStakingToken.withdraw()` is in the user's favor	Resolved
M-5	Deposits from mainnet to L2 can be forced to fail	Resolved
M-6	LayerZero `nativeFee` calculation in `_sendDeposit()` is wrong for `LiquidStakingTokenLockbox.depositEth()`	Resolved
L-1	pxEth approval to old `lockBox` is not revoked in `L1SyncPoolETH._setLockBox()`	Resolved

4 Findings

High Risk

[H-1] `DineroERC20RebaseUpgradeable._transferShares()` duplicates shares on self-transfer

Context: DineroERC20RebaseUpgradeable.sol#L305-L311

Description: `DineroERC20RebaseUpgradeable._transferShares()` is used to transfer shares from the `_sender` address to the `_recipient` address:

```
uint256 currentSenderShares = $.shares[_sender];
uint256 currentRecipientShares = $.shares[_recipient];

if (_shares > currentSenderShares) revert Errors.InvalidAmount();

$.shares[_sender] = currentSenderShares - _shares;
$.shares[_recipient] = currentRecipientShares + _shares;
```

However, since the recipient's balance is cached in `currentRecipientShares` before performing any calculation, if `_sender == _recipient` (ie. a user performs a self-transfer), his share balance will increase when it is not supposed to. As such, any user holding `DineroERC20RebaseUpgradeable` tokens can infinitely increase their balance by calling `transferShares()` to transfer shares to themselves.

Recommendation: Avoid caching the recipient's balance:

```
  uint256 currentSenderShares = $.shares[_sender];
- uint256 currentRecipientShares = $.shares[_recipient];

  if (_shares > currentSenderShares) revert Errors.InvalidAmount();

  $.shares[_sender] = currentSenderShares - _shares;
- $.shares[_recipient] = currentRecipientShares + _shares;
+ $.shares[_recipient] += _shares;
```

Redacted: Fixed in commit fb71390.

Renascence: Verified, the recommended fix was implemented. Additionally, a `_sender != recipient` check was added to prevent users from transferring to themselves.

[H-2] `LiquidStakingToken.mint()` doesnt add deposit fee shares to `RateLimiter.unsyncedLimit`

Context:

- LiquidStakingToken.sol#L307-L317
- LiquidStakingToken.sol#L326-L327

Description: In `LiquidStakingToken.mint()`, a portion of `shares` are minted to the treasury as the deposit fee:

```
uint256 depositFee = $.syncDepositFee;
if (depositFee > 0) {
    uint256 feeShares = shares.mulDivDown(
        depositFee,
        Constants.FEE_DENOMINATOR
    );

    _mintShares($.treasury, feeShares);

    shares -= feeShares;
}
```

Notice how `feeShares` is subtracted from `shares`, therefore, the leftover `shares` value is amount of shares minted to the user.

However, later on in the function, `RateLimiter.updateRateLimit()` is called with `shares`:

```
if (address($.rateLimiter) != address(0))
    $.rateLimiter.updateRateLimit(address(this), address(0), shares, 0);
```

As such, `RateLimiter.unsyncedLimit` is only updated with the amount of shares minted to the user. The shares minted to the treasury as the deposit fee is never added to `unsyncedLimit`. This makes `unsyncedLimit` lower than the actual amount of unsynced shares, which prevents some users from withdrawing their shares.

Recommendation: Call `updateRateLimit()` with `shares` before calculating the amount of shares belonging to the deposit fee:

```
+ if (address($.rateLimiter) != address(0))
+     $.rateLimiter.updateRateLimit(address(this), address(0), shares, 0);

  uint256 depositFee = $.syncDepositFee;
  if (depositFee > 0) {
      // ...
  }
```

Redacted: Fixed in commit 7c3718b.

Renascence: Verified, the recommended fix was implemented.

[H-3] `LiquidStakingToken.withdraw()` subtracts the amount of assets to withdraw from `_totalAssets()` twice

Context: LiquidStakingToken.sol#L382-L391

Description: In `LiquidStakingToken.withdraw()`, the amount of assets withdrawn (ie. `_amount`) is subtracted from `pendingDeposit` or `totalStaked` as shown:

```
if (pendingDeposit > 0) {
    if (pendingDeposit > _amount) {
        $.pendingDeposit -= _amount;
    } else {
        $.pendingDeposit = 0;
        _amount -= pendingDeposit;
    }
}

$.totalStaked -= _amount;
```

As seen from above, if `pendingDeposit` is non-zero, assets are subtracted from `pendingDeposit` before subtracting from `totalStaked`.

However, when `pendingDeposit > _amount`, `_amount` is not set to `0` after all assets withdrawn are subtracted from `pendingDeposit`. This causes `_amount` to be subtracted from `totalStaked` again, reducing `_totalAssets()` by double of the amount of assets withdrawn. As such, all users holding modeETH will have their balance reduced for no reason.

Recommendation: When `pendingDeposit > _amount`, set `_amount` to `0` to avoid subtracting from `totalStaked` afterwards:

```
  if (pendingDeposit > 0) {
      if (pendingDeposit > _amount) {
          $.pendingDeposit -= _amount;
+         _amount = 0;
      } else {
```

Redacted: Fixed in commit bb51717.

Renascence: Verified, the issue was fixed by only subtracting from `$.totalStaked` in the branches where `pendingDeposit` is insufficient to cover the full `_amount`.

[H-4] `LiquidStakingTokenLockbox._calculateWithdrawalAmount()` excluding `pendingDeposit` causes loss of funds on withdrawal

Context:

- LiquidStakingTokenLockbox.sol#L283-L288
- LiquidStakingTokenLockbox.sol#L688-L692

Description: When handling L2 withdrawals in `LiquidStakingTokenLockbox._lzReceive()`, `_calculateWithdrawalAmount()` is used to determine the amount of pxEth the user receives for his withdrawal:

```
// Calculate the amount to withdraw based on the latest assetsPerShare
// and assetsPerShare at the time of last successful rebase
uint256 postFeeAmount = _calculateWithdrawalAmount(
    _amount,
    _assetsPerShare
);
```

However, `_calculateWithdrawalAmount()` limits the amount of pxEth that can be withdrawn pxEth value of the apxEth held in the contract, excluding `pendingDeposit`:

```
uint256 totalAssets = autoPxEth.previewRedeem(
    autoPxEth.balanceOf(address(this))
);

return previewRedeem > totalAssets ? totalAssets : previewRedeem;
```

As such, if the amount of apxEth held excluding `pendingDeposit` is insufficient to cover the user's withdrawal, the user will lose funds. For example:

- Assume the protocol is new and has no deposits.
- Bob calls `L2BaseSyncPoolUpgradeable.deposit()` to deposit 10 ETH for modeETH.
- Bob calls `L2BaseSyncPoolUpgradeable.sync()` to send the 10 ETH to mainnet. This sends two messages to L1:
  - A fast message through LayerZero that calls `L1SyncPoolETH._anticipatedDeposit()`.
  - A slow message through the native L2 bridge (which typically takes 7 days) that calls `_finalizeDeposit()`.
- The fast message reaches first, calling `_anticipatedDeposit()` and adding 10 ETH to `pendingDeposit` in `depositSync()`.
- Bob calls `LiquidStakingToken.withdraw()` to withdraw his 10 ETH.
- `LiquidStakingTokenLockbox._lzReceive()` receives the message to handle the withdrawal on L1:
  - Since there is no apxEth held in the contract yet, `_calculateWithdrawalAmount()` returns `0`.
  - Bob receives nothing for his withdrawal.

in this example, since `_calculateWithdrawalAmount()` does not include `pendingDeposit`, Bob received no funds on withdrawal even though there was sufficient ETH in `pendingDeposit`.

Recommendation: In `_calculateWithdrawalAmount()`, add `pendingDeposit` to `totalAssets`:

```
   uint256 totalAssets = autoPxEth.previewRedeem(
       autoPxEth.balanceOf(address(this))
 - );
 + ) + $.pendingDeposit;

   return previewRedeem > totalAssets ? totalAssets : previewRedeem;
```

Redacted: Fixed in commit ec96678.

Renascence: Verified, the recommended fix was implemented.

[H-5] Incorrect check in `RateLimiter.updateRateLimit()` causes `unsyncedLimit` balance to be frozen

Context:

- RateLimiter.sol#L56-L59
- L2ModeSyncPoolETH.sol#L156-L161

Description: In `RateLimiter.updateRateLimit()`, funds are moved from the `unsyncedLimit` to the `withdrawLimit` under the following if-condition:

```
} else if (msg.sender == syncPool && token == syncPool) {
    withdrawLimit += unsyncedLimit;
    unsyncedLimit = 0;
}
```

However, this if-condition will never trigger as `syncPool` is not a token. In `L2ModeSyncPoolETH._sync()`, `updateRateLimit()` is actually called with `token` as `address(0)`:

```
IRateLimiter(getRateLimiter()).updateRateLimit(
    address(this),
    address(0),
    amountIn,
    amountOut
);
```

As such, modeETH minted on L2 will forever remain in the `unsyncedLimited` balance, which causes withdrawals to be permanently blocked.

Recommendation: In `RateLimiter.updateRateLimit()`, check if `token == address(0)` instead:

```diff
- } else if (msg.sender == syncPool && token == syncPool) {
+ } else if (msg.sender == syncPool && token == address(0)) {
      withdrawLimit += unsyncedLimit;
      unsyncedLimit = 0;
  }
```

Redacted: Fixed in commit 172df42.

Renascence: Verified, the issue was fixed by calling `updateRateLimit()` with `token = address(this)` instead.

[H-6] Total assets in `LiquidStakingToken` is wrongly reduced twice for a withdrawal

Context:

- LiquidStakingToken.sol#L380-L391
- LiquidStakingTokenLockbox.sol#L293-L313
- LiquidStakingToken.sol#L281-L283

Description: When `LiquidStakingToken.withdraw()` is called to withdraw modeETH, the amount withdrawn is subtracted from `pendingDeposit` and `totalStaked`:

```
uint256 pendingDeposit = $.pendingDeposit;

if (pendingDeposit > 0) {
    if (pendingDeposit > _amount) {
        $.pendingDeposit -= _amount;
    } else {
        $.pendingDeposit = 0;
        _amount -= pendingDeposit;
    }
}

$.totalStaked -= _amount;
```

Subsequently, when `LiquidStakingTokenLockbox._lzReceive()` is called to handle the withdrawal on L1, the amount withdrawn is added to `pendingWithdraw`:

```
if (pendingAmount > 0) {
    if (pendingAmount >= postFeeAmount) {
        $.pendingDeposit -= postFeeAmount;

        $.pendingWithdraw += postFeeAmount;

        $.pxEth.safeTransfer(_receiver, postFeeAmount);

        emit Withdrawal(_guid, _receiver, postFeeAmount);

        return;
    }

    $.pxEth.safeTransfer(_receiver, pendingAmount);

    $.pendingDeposit = 0;

    postFeeAmount -= pendingAmount;

    $.pendingWithdraw += postFeeAmount;
}
```

When `rebase()` or any deposit function in `LiquidStakingTokenLockbox` is called, this ends up subtracting from `pendingDeposit` in `LiquidStakingToken` again:

```
_pendingWithdraw > $.pendingDeposit
    ? $.pendingDeposit = 0
    : $.pendingDeposit -= _pendingWithdraw;
```

As such, whenever a withdrawal occurs, the amount withdrawn is subtracted from `pendingDeposit` twice. This reduces `_totalAssets()` in `LiquidStakingToken` by more than the withdrawn amount, causing a loss of funds for all users holding modeETH.

Recommendation: In `LiquidStakingTokenLockbox._lzReceive()`, the amount withdrawn should not be added to `pendingWithdraw`. This prevents subtracting from `pendingDeposit` for a second time.

Redacted: Fixed in commit 173a3a4.

Renascence: Verified, `pendingWithdraw` is no longer increased in `LiquidStakingTokenLockbox._lzReceive()`.

[H-7] totalStaked is not increased for finalized L2 deposits

Context: LiquidStakingToken.sol#L281-L283

Description: An L2 deposit for modeETH has the following transaction lifecycle. On L2:

- User calls L2BaseSyncPoolUpgradeable.deposit() to deposit ETH for modeETH:
  - LiquidStakingToken.mint() is called, which adds the deposited ETH to pendingDeposit.
- User calls L2BaseSyncPoolUpgradeable.sync() to send the ETH to mainnet. This sends two messages to L1:
  - A fast message through LayerZero that calls L1SyncPoolETH._anticipatedDeposit().
  - A slow message through the native L2 bridge (which typically takes 7 days) that calls L1SyncPoolETH._finalizeDeposit().

Afterwards, on L1:

- The fast message reaches L1 first, calling _anticipatedDeposit():
  - LiquidStakingTokenLockbox.depositSync() is called, which adds ETH to pendingDeposit and mints pxEth.
- The slow message reaches L1 later, calling _finalizedDeposit(). LiquidStakingTokenLockbox.depositSync() is called, which does the following:
  - Removes ETH from pendingDeposit and adds it to pendingWithdraw.
  - Deposits pxEth for apxEth.

Subsequently, when rebase() or any deposit function in LiquidStakingTokenLockbox is called, the value of pendingWithdraw will be sent to L2 and used to subtract from pendingDeposit on L2 in LiquidStakingToken._lzReceive():

```
_pendingWithdraw > $.pendingDeposit
    ? $.pendingDeposit = 0
    : $.pendingDeposit -= _pendingWithdraw;
```

However, the subtracted amount is never added to totalStaked, even though the pending pxEth has already been deposited for apxEth on L1. Since _totalAssets() is the sum of totalStaked and pendingDeposit, its value will be reduced after a L2 deposit is finalized, causing a loss of funds for all modeETH holders.

Recommendation: In LiquidStakingToken._lzReceive(), the amount subtracted from pendingDeposit should be added to totalStaked:

```
- _pendingWithdraw > $.pendingDeposit
-     ? $.pendingDeposit = 0
-     : $.pendingDeposit -= _pendingWithdraw;
+ uint256 pendingDeposit = $.pendingDeposit;
+ uint256 delta = _pendingWithdraw > pendingDeposit ? pendingDeposit :
_pendingWithdraw;
+ $.pendingDeposit -= delta;
+ $.totalStaked += delta;
```

Note that this fix requires `pendingWithdraw` to only be increased when a deposit is finalized. `pendingWithdraw` must not increase when handling a L2 withdrawal in `LiquidStakingTokenLockbox._lzReceive()`.

Redacted: Fixed in commit ad5c5ee.

Renascence: The fix calls `_updateTotalStaked(_pendingWithdraw, _assetsPerShare)` to add `_pendingWithdraw` to `$.totalStaked`:

```
if (_pendingWithdraw > 0) {
    _pendingWithdraw > $.pendingDeposit
        ? $.pendingDeposit = 0
        : $.pendingDeposit -= _pendingWithdraw;

    _updateTotalStaked(_pendingWithdraw, _assetsPerShare);
}

// update last assets per share
$.lastAssetsPerShare = _assetsPerShare;
```

The issue is that earlier in the function, `_updateTotalStaked()` was already called and `$.totalStaked` has been scaled by `_assetsPerShare / $.lastAssetsPerShare`. However, since `$.lastAssetsPerShare` is only updated at the end of the function, when `_updateTotalStaked()` is called again here, `$.totalStaked` will be scaled again, resulting in its value being scaled one extra time.

Consider adding to `$.totalStaked` directly:

```diff
  if (_pendingWithdraw > 0) {
      _pendingWithdraw > $.pendingDeposit
          ? $.pendingDeposit = 0
          : $.pendingDeposit -= _pendingWithdraw;

-     _updateTotalStaked(_pendingWithdraw, _assetsPerShare);
+     $.totalStaked += _pendingWithdraw;
  }
```

Redacted: Amended in commit beebc32.

Renascence: Verified, the function now adds to `$.totalStaked` directly instead of calling `_updateTotalStaked()` twice.

Medium Risk

[M-1] `L1SyncPoolETH._finalizeDeposit()` does not account for the `PirexEth` deposit fee

Context:

- L1SyncPoolETH.sol#L155-L162
- PirexEth.sol#L307-L314

Description: When `L1SyncPoolETH._finalizeDeposit()` is called, it calls `PirexEth.deposit()` to deposit ETH for pxEth. Afterwards, the pxEth is burnt, and the previously minted pxEth in `LiquidStakingTokenLockbox` is staked for apxEth:

```
// sent the ETH to PirexETH
IPirexEth(getPlatform()).deposit{value: amountIn}(address(this), false);

// burn the pxEth
IDineroERC20(getTokenOut()).burn(address(this), amountIn);

// notify the lockbox to deposit up to amountIn into the vault
LiquidStakingTokenLockbox(getLockBox()).depositSync(amountIn, true);
```

As seen from above, the function assumes that the amount of pxEth received from `PirexEth.deposit()` is always `amountIn`. However, `PirexEth.deposit()` charges a deposit fee:

```
// Get the pxETH amounts for the receiver and the protocol (fees)
(postFeeAmount, feeAmount) = _computeAssetAmounts(
    DataTypes.Fees.Deposit,
    msg.value
);

// Mint pxETH for the receiver (or this contract if compounding) excluding fees
_mintPxEth(shouldCompound ? address(this) : receiver, postFeeAmount);
```

Due to the deposit fee, the amount of pxEth received from `PirexEth.deposit()` in `L1SyncPoolETH._finalizeDeposit()` is equal to `postFeeAmount`, which is less than `amountIn`. As such, when `_finalizeDeposit()` is called while the `PirexEth` deposit fee is active, it will revert due to an insufficient pxEth balance.

Recommendation: In `L1SyncPoolETH._finalizeDeposit()`, burn `postFeeAmount` of pxEth instead of `amountIn`:

```diff
  // sent the ETH to PirexETH
- IPirexEth(getPlatform()).deposit{value: amountIn}(address(this), false);
+ (
+     uint256 postFeeAmount,
+     uint256 feeAmount
+ ) = IPirexEth(getPlatform()).deposit{value: amountIn}(address(this), false);

  // burn the pxEth
- IDineroERC20(getTokenOut()).burn(address(this), amountIn);
+ IDineroERC20(getTokenOut()).burn(address(this), postFeeAmount);

  // notify the lockbox to deposit up to amountIn into the vault
  LiquidStakingTokenLockbox(getLockBox()).depositSync(amountIn, true);
```

Additionally, when `depositSync()` is called, an amount of pxEth equal to `feeAmount` should be burned from `LiquidStakingTokenLockbox`.

Redacted: Fixed in commit b9a7cb4.

Renascence: This fix is incomplete - `feeAmount` worth of pxEth still needs to be burn from `LiquidStakingTokenLockbox` when `depositSync()` is called.

When `depositSync()` is called to resolve pending deposits, `postFeeAmount` of pxEth should be deposited into apxEth, while the remaining `feeAmount` of pxEth should be burnt. Otherwise, there's an extra `feeAmount` of pxEth in circulation whenever an L2 → L1 deposit occurs. `pendingDeposit` in `LiquidStakingTokenLockbox` will also be inflated.

Redacted: Amended in commit be1b483.

It's important to clarify that we never intend to set a `depositFee`. However, if a change to the `depositFee` occurs, the oracle must inform L2 on every syncpool deposit about the expected `postFeeAmount` of pxEth. This value is sent to L1 as `amountOut`.

Changes Made:

- `_anticipatedDeposit` now mints `amountOut` instead of `amountIn`.
- `_finalizeDeposit` now burns the smaller amount between `postFeeAmount` and `amountOut`.

Scenarios:

- If L2 mints the correct pxEth amount considering the `depositFee`, everything works as expected.
- If the `depositFee` is reduced while the final transaction is in transit, L2 mints less than what `L1SyncPoolETH` will receive. To address this, a function is added allowing the owner to withdraw the extra tokens.
- If the `depositFee` is increased while the final transaction is in transit, L2 mints more than what `L1SyncPoolETH` will receive. In this case, `L1SyncPoolETH` will burn the entire `postFeeAmount`. The excess pxEth received will be managed by the treasury, which will burn the extra tokens to ensure the lockbox is backed by the minted amount. Since increasing the `depositFee` is a very unlikely scenario, this workaround is considered a sufficient solution.

Redacted: Verified. As long as the oracle accurately reports the expected `postFeeAmount` of pxEth, the amount of pxEth minted in `_anticipatedDeposit()` will match the amount received from depositing ETH in `_finalizeDeposit()`.

[M-2] `LiquidStakingToken` and `LiquidStakingTokenLockbox` are not pausable

Context:

- LiquidStakingToken.sol#L27
- LiquidStakingTokenLockbox.sol#L28

Description: Both `LiquidStakingToken` and `LiquidStakingTokenLockbox` inherit `PausableUpgradeable` so that the owner has the ability to pause functions with the `whenNotPaused` modifier when needed. However, both contracts do not contain any functions that call `_pause()` or `_unpause()`, and as such, both contracts are actually not pausable.

Recommendation: Add permissioned functions to both contracts that call `_pause()` and `_unpause()`.

Redacted: Fixed in commit d1fdf02.

Renascence: Verified, the recommended fix was implemented.

[M-3] Depositors unfairly accrue yield from the `assetsPerShare` difference when minting mod-eETH

Context:

- LiquidStakingToken.sol#L304-L324
- LiquidStakingToken.sol#L557-L561
- LiquidStakingToken.sol#L229-L235

Description: When `LiquidStakingToken.mint()` is called, it calculates the amount of shares to be minted to the depositor before calling `_updateTotalStaked()`:

```
uint256 _totalShares = getTotalShares();
uint256 shares = _totalShares == 0 ? _amount : convertToShares(_amount);

// ...

_mintShares(_to, shares);

_updateTotalStaked(0, _assetsPerShare);

$.pendingDeposit += _amount;
$.lastAssetsPerShare = _assetsPerShare;
```

`_updateTotalStaked()` increases `totalStaked` based on the difference between the last recorded `assetsPerShare` and the current `assetsPerShare`:

```
_lastAssetsPerShare == 0
    ? $.totalStaked = _totalStaked + _amount
    : $.totalStaked =
    _totalStaked.mulDivDown(_assetsPerShare, _lastAssetsPerShare) +
    _amount;
```

However, since `_totalAssets()` is the sum of `totalStaked` and `pendingDeposit`, calculating the amount of shares to mint before calling `_updateTotalStaked()` unfairly accrues a portion of yield to the caller, even though he just deposited. This is because `_totalAssets()` is only increased after the amount of shares to mint is calculated.

For example:

- Assume the following:
  - The deposit fee is 0%.
  - `pendingDeposit = 0` and `totalStaked = 10 ether`, so `_totalAssets() = 10 ether`
  - `_totalShares = 10e18`
  - `_lastAssetsPerShare = 1e18`
- A user calls `mint()` with 10 ETH, and the current `_assetsPerShare` is `2e18`:
  - The user receives `10e18` shares as `shares = 10 ether * 10e18 / 10 ether = 10e18`
  - `totalShares` increases to `20e18`.
  - In `_updateTotalStaked()`:
    - `totalStaked = 10 ether * 2e18 / 1e18 = 20 ether`
  - The user's 10 ETH is added to `pendingDeposit`, so `pendingDeposit = 10 ether`
  - Now ,`_totalAssets() = 10 ether + 20 ether = 30 ether`
- If the user withdraws immediately, his `10e18` shares are worth half of `_totalAssets()`, which is 15 ETH.
- However, he only deposited 10 ETH. As such, he has gained 5 ETH of yield unfairly.

Note that `_updateTotalStaked()` is also called after share calculation in `LiquidStakingToken._lzReceive()` when handling L1 deposits, so the same bug occurs there as well:

```
uint256 shares = _totalShares == 0
    ? _amount
    : convertToShares(_amount);

_mintShares(_receiver, shares);

_updateTotalStaked(_amount, _assetsPerShare);
```

Recommendation: In `LiquidStakingToken.mint()`, call `_updateTotalStaked()` to update `totalStaked` before calculating the amount of shares to mint:

```
+ _updateTotalStaked(0, _assetsPerShare);

  uint256 _totalShares = getTotalShares();
  uint256 shares = _totalShares == 0 ? _amount : convertToShares(_amount);

  // ...

  _mintShares(_to, shares);

- _updateTotalStaked(0, _assetsPerShare);
```

The same can be done in `LiquidStakingToken._lzReceive()`:

```
+ _updateTotalStaked(0, _assetsPerShare);

  uint256 shares = _totalShares == 0
      ? _amount
      : convertToShares(_amount);

  _mintShares(_receiver, shares);

- _updateTotalStaked(_amount, _assetsPerShare);
+ $.totalStaked += _amount;
```

Note that the newly deposited `_amount` should only be added to `totalStaked` after share calculation is performed.

Redacted: Fixed in commit aaa001a.

Renascence: Verified, the recommended fix was implemented.

[M-4] Rounding for share calculation in `LiquidStakingToken.withdraw()` is in the users favor

Context:

- LiquidStakingToken.sol#L351
- DineroERC20RebaseUpgradeable.sol#L257-L265
- LiquidStakingToken.sol#L391

Description: When users call `withdraw()` with the amount of assets to withdraw (ie. `_amount`), the amount of shares to burn is calculated using `convertedToShares()`:

```
uint256 shares = convertToShares(_amount);
```

However, `convertToShares()` rounds down the number of shares calculated:

```
function convertToShares(uint256 _assets) public view returns (uint256) {
    uint256 totalShares = _getDineroERC20RebaseStorage().totalShares;
    uint256 totalPooledPxEth = _totalAssets();

    return
        totalPooledPxEth == 0
            ? 0
            : _assets.mulDivDown(totalShares, totalPooledPxEth);
}
```

Since the amount of shares to burn in `withdraw()` is rounded down, it is in the user's favor and against the protocol. If `_totalAssets()` is relatively large compared to `totalShares`, this becomes dangerous as `_amount * totalShares / _totalAssets()` could round down to 0, allowing the user to withdraw a non-trivial amount of assets for no shares.

Additionally, this makes it possible for an underflow to occur when subtracting `_amount` from `totalStaked`:

```
$.totalStaked -= _amount;
```

For example:

- Assume the following:
  - There is only one user with 10 ETH staked and `1e18` shares.
  - This means `totalStaked = 10 ether` and `_totalShares = 1e18`.
- He calls `withdraw()` with `_amount = 10 ether + 1`:
  - `shares = (10 ether + 1) * 1e18 / 10 ether` rounds down to `1e18`.
  - `totalStaked - _amount = 10 ether - (10 ether + 1)`, so the calculation mentioned above reverts.

When there is more than one user holding modeETH, this rounding error in `withdraw()` will cause `totalStaked` to become smaller than the actual amount of ETH staked. Eventually, when the last user attempts to withdraw his modeETH, `withdraw()` will revert.

Recommendation: In `withdraw()`, the amount of shares to burn from the user should be rounded up:

```
- uint256 shares = convertToShares(_amount);
+ uint256 totalShares = _getDineroERC20RebaseStorage().totalShares;
+ uint256 totalAssets = _totalAssets();
+ uint256 shares = totalAssets == 0 ? 0 : _amount.mulDivUp(totalShares, totalAssets);
```

Redacted: Fixed in commit af4ca5c.

Renascence: This fix appears simple but affects other parts of the code.

The `_update()` hook uses `convertToShares()` to calculate how many shares to transfer when users transfer their balance:

```
function _update(
    address _sender,
    address _recipient,
    uint256 _amount
) internal override {
    uint256 sharesToTransfer = convertToShares(_amount);
    _transferShares(_sender, _recipient, sharesToTransfer);
    _emitTransferEvents(_sender, _recipient, _amount, sharesToTransfer);
}
```

Since `convertToShares()` now rounds up, the calculated `sharesToTransfer` could actually end up becoming larger than the user's actual share balance, causing regular balance transfers to revert.

`WrappedLiquidStakedToken .wrap()` uses `convertToShares()` to calculate the amount of shares to mint for LST assets:

```
uint256 shares = lst.convertToShares(_amount);

_mint(msg.sender, shares);
```

With the new change, this now rounds against the protocol as rounding up gives the user more shares.

Only the calculation in `LiquidStakingToken.withdraw()` should round up, so it's not feasible to just change `convertToShares()` directly. Consider either adding a parameter to `convertToShares()` to specify the rounding direction, or an entirely new function that rounds up the share calculation.

Redacted: Amended in commit 98bcb6a, added an extra view function named `previewWithdraw()` for rounding up.

Renascence: Verified, only `LiquidStakingToken.withdraw()` rounds up now.

[M-5] Deposits from mainnet to L2 can be forced to fail

Context:

- DineroERC20RebaseUpgradeable.sol#L325-L329
- LiquidStakingTokenLockbox.sol#L363-L373

Description: When `DineroERC20RebaseUpgradeable._mintShares()` is called with `_recipient` as the zero address, it reverts:

```
function _mintShares(
    address _recipient,
    uint256 _shares
) internal returns (uint256) {
    if (_recipient == address(0)) revert Errors.ZeroAddress();
```

`_mintShares()` is called in `LiquidStakingToken._lzReceive()` when handling L1 deposits.

However, the issue is that `depositEth()`, `depositPxEth()` and `depositApxEth()` in `LiquidStakingTokenLockbox` do not ensure that the `_receiver` specified by the user is not the zero address. Therefore, a user can call any of the three functions with `_receiver = address(0)`, and `LiquidStakingToken._lzReceive()` is guaranteed to always revert when handling the message sent to L2.

This is problematic as crucial information is sent along with L1 → L2 deposit messages, such as `pendingWithdraw`:

```
_sendDeposit(
    postFeeAmount,
    assetsPerShare,
    $.pendingWithdraw,
    _receiver,
    _refundAddress,
    address(0),
    _options
);

$.pendingWithdraw = 0;
```

`pendingWithdraw` is the amount of pending deposits have been staked, and can be moved from `pendingDeposit` to `totalStaked` on the L2 side. However, if a user calls any L1 deposit function with `_receiver == address(0)`, `pendingWithdraw` will be reset to `0` on L1, but the message on L2 will never be processed. This causes a portion of deposits to forever be stuck in `pendingDeposit`, resulting in a loss of yield for modeETH holders.

Recommendation: In `depositEth()`, `depositPxEth()` and `depositApxEth()`, check that `_receiver` is not the zero address and revert if so.

Redacted: Fixed in commit 1346333.

Renascence: Verified, the `_receiver == address(0)` check was added to all deposit functions in `LiquidStakingTokenLockbox`.

[M-6] LayerZero `nativeFee` calculation in `_sendDeposit()` is wrong for `LiquidStakingTokenLockbox.depositEth()`

Context:

- LiquidStakingTokenLockbox.sol#L349-L371
- LiquidStakingTokenLockbox.sol#L743-L745

Description: In `LiquidStakingTokenLockbox.depositEth()`, the ETH sent (ie. `msg.value`) includes both the amount of ETH sent to `PirexEth` for deposit (ie. `_amount`) and ETH for LayerZero fees (ie. `nativeFee`).

After ETH is deposited into `PirexEth`, `_sendDeposit()` is called to send a LayerZero message with `postFeeAmount`:

```
// Deposit via PirexEth and receive apxEth in return to be kept in this vault
(uint256 postFeeAmount, ) = $.pirexEth.deposit{value: _amount}(
    address(this),
    true
);

// ...

_sendDeposit(
    postFeeAmount,
    // ...
);
```

`nativeFee` is then calculated as `msg.value - postFeeAmount` in `_sendDeposit()` (note that `_amount` below is `postFeeAmount`):

```
uint256 nativeFee = _asset == address(0)
    ? msg.value - _amount
    : msg.value;
```

This is incorrect as `msg.value - postFeeAmount` includes the deposit fee taken by `PirexEth.deposit()`. The correct calculation for `nativeFee` would be `nativeFee = msg.value - postFeeAmount - feeAmount`.

As such, if `PirexEth.deposit()` charges a deposit fee, `depositEth()` will revert due to insufficient ETH when trying to send `nativeFee` to LayerZero.

Recommendation: Consider adding a `feeAmount` parameter to `_sendDeposit`, which is the deposit fee amount taken by `PirexEth.deposit()`:

```
  // Deposit via PirexEth and receive apxEth in return to be kept in this vault
- (uint256 postFeeAmount, ) = $.pirexEth.deposit{value: _amount}(
+ (uint256 postFeeAmount, uint256 feeAmount) = $.pirexEth.deposit{value: _amount}(
      address(this),
      true
  );

  // ...

  _sendDeposit(
      postFeeAmount,
+     feeAmount,
      // ...
  );
```

`nativeFee` can then be calculated as such:

```
  uint256 nativeFee = _asset == address(0)
-     ? msg.value - _amount
+     ? msg.value - _amount - feeAmount
      : msg.value;
```

Redacted: Fixed in commit 52f8de1.

Renascence: Verified. The native fee amount is now calculated in their respective deposit functions and passed to `_sendDeposit()` directly.

Low Risk

[L-1] pxEth approval to old `lockBox` is not revoked in `L1SyncPoolETH._setLockBox()`

Context: L1SyncPoolETH.sol#L169-L176

Description: In `L1SyncPoolETH`, the owner can call `setLockBox()` to change the `lockBox` address. This grants pxEth approval to the new `lockBox` address:

```
function _setLockBox(address lockBox) internal override {
    L1BaseSyncPoolStorage storage $ = _getL1BaseSyncPoolStorage();
    $.lockBox = lockBox;

    IDineroERC20(getTokenOut()).approve(lockBox, type(uint256).max);

    emit LockBoxSet(lockBox);
}
```

However, the approval to the previous `lockBox` address is not revoked. This could be dangerous if the approval to the old `lockBox` can be manipulated by attackers.

Recommendation: Revoke approval to the old `lockBox` if it was set:

```
function _setLockBox(address lockBox) internal override {
    L1BaseSyncPoolStorage storage $ = _getL1BaseSyncPoolStorage();

    if ($.lockBox != address(0)) IDineroERC20(getTokenOut()).approve($.lockBox, 0);

    $.lockBox = lockBox;
    IDineroERC20(getTokenOut()).approve(lockBox, type(uint256).max);

    emit LockBoxSet(lockBox);
}
```

Redacted: Fixed in commit a946b38.

Renascence: Verified, the recommended fix was implemented.