RENASCENCE

Dinero (Branded LST) Audit Report

Version 2.0

Audited by:

MiloTruck

bytes032

August 12, 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	55207ef5b814...
Mitigation Hash	0d0f3c896fcc...
Date	5 August 2024 - 9 August 2024

2.3 Issues Found

Severity	**Count**
High Risk	2
Medium Risk	2
Low Risk	4
Informational	0
Total Issues	**8**

3 Findings Summary

ID	Description	Status
H-1	Incorrect share calculation for rebase fee in `LiquidStakingToken._lzReceive()`	Resolved
H-2	Reentrancy attacks through `_lzSend()` refunds	Resolved
M-1	Allowance calculation in `DineroERC20RebaseUpgradeable.transferSharesFrom()` rounds down in the spender's favor	Resolved
M-2	Fluctuations in share price affect how much `withdrawLimit` is increased by in `LiquidStakingToken._sync()`	Resolved
L-1	`L1SyncPool.withdraw()` also transfers unbacked pxEth out	Resolved
L-2	Inefficient queue implementation for `syncedIds` in `LiquidStakingTokenLockbox`	Resolved
L-3	`lastCompletedSyncIndex` is not updated in `LiquidStakingToken._withdrawPendingDeposit()`	Resolved
L-4	Missing `nonReentrant` modifier on user-facing functions	Resolved

4 Findings

High Risk

[H-1] Incorrect share calculation for rebase fee in `LiquidStakingToken._lzReceive()`

Context: LiquidStakingToken.sol#L298-L300

Description: In `LiquidStakingToken._lzReceive()`, the amount of shares to mint to the treasury for the rebase fee is calculated as such:

```
shares = fee.mulDivDown(getTotalShares(), _totalAssets());

_mintShares($.treasury, shares);
```

Note that `fee` is the amount of assets that the treasury should receive.

However, dividing by `_totalAssets()` is incorrect and will cause the treasury to receive less assets than intended. For example:

- Assume that `totalShares = 100e18` and `_totalAssets() = 100e18`.
- If `fee = 20e18`, the following logic occurs:
  - `shares = 20e18 * 100e18 / 100e18 = 20e18`.
  - `_mintShares()` adds `shares` to `totalShares`, so `totalShares` is updated to `100e18 + 20e18 = 120e18`.
- The amount of assets received by the treasury is:
  - `shares * _totalAssets() / totalShares = 20e18 * 100e18 / 120e18 = ~16e18`.

Instead of `20e18` assets, the treasury only receives `~16e18` assets from the rebase fee.

Recommendation: When calculating the amount of shares to mint, divide by `_totalAssets() - fee` instead:

```
- shares = fee.mulDivDown(getTotalShares(), _totalAssets());
+ shares = fee.mulDivDown(getTotalShares(), _totalAssets() - fee);
```

Now, when mint shares for `fee = 20e18`, the following logic occurs:

- `shares = 20e18 * 100e18 / (100e18 - 20e18) = 25e18`.
- `totalShares = 100e18 + 25e18 = 125e18`.
- The amount of assets received by the treasury is:
  - `shares * _totalAssets() / totalShares = 25e18 * 100e18 / 125e18 = 20e18`.

Redacted: Fixed in commit d1722e2.

Renascence: Verified, the recommended fix was implemented.

[H-2] Reentrancy attacks through `_lzSend()` refunds

Context:

- LiquidStakingToken.sol#L432-L438
- LiquidStakingToken.sol#L813-L824
- LiquidStakingTokenLockbox.sol#L485-L491
- LiquidStakingTokenLockbox.sol#L780-L786

Description: Throughout the protocol, functions that call `_lzSend()` to send a cross-chain message allow the caller to specify the native fee amount and the address that receives refunds.

Using `LiquidStakingToken.withdraw()` as an example, the native fee sent to LayerZero's endpoint is `msg.value`, which is not validated, and `_refundAddress` is specified by the caller:

```
MessagingReceipt memory msgReceipt = _lzSend(
    L1_EID,
    payload,
    combinedOptions,
    MessagingFee(msg.value, 0),
    payable(_refundAddress)
);
```

In LayerZero's endpoint, when `send()` is called with `msg.value` more than the required native fee amount, the excess ETH is sent back to `_refundAddress`, as seen below.

In `EndpointV2.send()`:

```
// handle native fees
_payNative(receipt.fee.nativeFee, suppliedNative, _sendLibrary, _refundAddress);
```

In `EndpointV2._payNative()`:

```
if (_required < _supplied) {
    unchecked {
        // refund the excess
        Transfer.native(_refundAddress, _supplied - _required);
    }
}
```

Therefore, if a user calls `LiquidStakingToken.withdraw()` and overpays for the LayerZero native fee, he will receive a callback when the `_refundAddress` receives ETH. This is true for all functions that call `_lzSend()` in the protocol.

However, in the current implementation of the protocol, this is dangerous as control flow is transferred to the caller mid-execution, allowing the caller to perform reentrancy attacks.

For example, `LiquidStakingToken.withdraw()` calls `_lzSend()` before subtracting from `totalStaked`:

```
MessagingReceipt memory msgReceipt = _lzSend(
    // ...
);

uint256 synced = $.syncedPendingDeposit;

if (synced > 0) {
    uint256 remaining = _withdrawPendingDeposit(_amount);
    if (remaining > 0) {
        $.totalStaked -= remaining;
    }
} else {
    $.totalStaked -= _amount;
}
```

An attacker can abuse reentrancy to exploit this as such:

- Call `LiquidStakingToken.withdraw()` with excess ETH:
  - Shares are burnt from the attacker and `totalShares` is decreased.
  - `_lzSend()` calls `EndpointV2.send()`, which sends ETH to `_refundAddress()`.
  - In the callback, call `EndpointV2.lzReceive()` to trigger `LiquidStakingToken._lzReceive()`:
    - Assume that `_lzReceive()` processes a `MESSAGE_TYPE_DEPOSIT` message for a victim.
    - Since `totalShares` has been decreased but not `totalStaked`, the victim's deposit is processed at an inflated share price, giving him less shares.
  - After `_lzSend()`, `totalStaked` is then decreased.

As seen from above, an attacker can re-enter `LiquidStakingToken._lzReceive()` to cause a user to receive less shares for an L1 deposit.

Note that this is only one of the possible attacks enabled by the ETH refund in `_lzSend()`. Reentrancy can be abused in all functions that call `_lzSend()`, more specifically:

- `LiquidStakingToken.withdraw()`
- `LiquidStakingToken._sync()`
- `LiquidStakingTokenLockbox.rebase()`
- `LiquidStakingTokenLockbox._sendDeposit()`

Recommendation: As recommended in **L-04**, add the `nonReentrant` modifier to all user-facing functions.

Additionally, in the following functions, call `_lzSend()` after all state updates have been performed:

- `LiquidStakingToken.withdraw()`

```
- MessagingReceipt memory msgReceipt = _lzSend(
-     // ...
- );

  uint256 synced = $.syncedPendingDeposit;

  if (synced > 0) {
      uint256 remaining = _withdrawPendingDeposit(_amount);
      if (remaining > 0) {
          $.totalStaked -= remaining;
      }
  } else {
      $.totalStaked -= _amount;
  }

+ MessagingReceipt memory msgReceipt = _lzSend(
+     // ...
+ );
```

- `LiquidStakingTokenLockbox.rebase()`

```
+ $.avgAssetsPerShare = assetsPerShare;
+ _updateSyncedIds();

  MessagingReceipt memory msgReceipt = _lzSend(
      // ...
  );

- $.avgAssetsPerShare = assetsPerShare;
- _updateSyncedIds();
```

- `LiquidStakingTokenLockbox._sendDeposit()`

```
+ _updateSyncedIds();

  MessagingReceipt memory msgReceipt = _lzSend(
      // ...
  );

- _updateSyncedIds();
```

Redacted: Fixed in commit 0d0f3c8.

Renascence: Verified, the recommended fix was implemented. Note that the recommended change in `LiquidStakingToken._sendDeposit()` was not made as the call to `_updateSyncedIds()` was removed in a later commit.

Medium Risk

[M-1] Allowance calculation in `DineroERC20RebaseUpgradeable.transferSharesFrom()` rounds down in the spenders favor

Context:

- DineroERC20RebaseUpgradeable.sol#L229-L230
- DineroERC20RebaseUpgradeable.sol#L253-L260

Description: When calling `DineroERC20RebaseUpgradeable.transferSharesFrom()`, the spender specifies the amount of `_shares` to be transferred from the owner. The allowance to deduct is then calculated with `convertToAssets()`:

```
uint256 assets = convertToAssets(_shares);
_spendAllowance(_sender, msg.sender, assets);
```

However, `convertToAssets()` rounds down:

```
function convertToAssets(uint256 _shares) public view returns (uint256) {
    uint256 totalShares = _getDineroERC20RebaseStorage().totalShares;

    return
        totalShares == 0
            ? 0
            : _shares.mulDivDown(_totalAssets(), totalShares);
}
```

Therefore, the amount of allowance to deduct rounds down in the spender's favor. If `_totalAssets()` is relatively smaller compared to `totalShares`, this becomes dangerous as `_shares * _totalAssets() / totalShares` could round down to `0`, allowing a spender to transfer a non-trivial amount of shares without any allowance.

For example:

- Assume that `_totalAssets() = 1e18` and `totalShares = 1e36 + 1`
- Attacker calls `transferSharesFrom()` with `_shares = 1e18`:
  - `_shares * _totalAssets() / totalShares = 1e18 * 1e18 / (1e36 + 1)` rounds down to `0`
  - `1e18` shares are transferred, but no allowance is deducted.

Recommendation: Calculate the allowance to subtract with `mulDivUp()` instead:

```
- uint256 assets = convertToAssets(_shares);
+ uint256 totalShares = _getDineroERC20RebaseStorage().totalShares;
+ uint256 assets = _shares.mulDivUp(_totalAssets(), totalShares);
  _spendAllowance(_sender, msg.sender, assets);
```

Redacted: Fixed in commit 4d87ba1.

Renascence: Verified. `convertToAssets()` has been modified to take in a `floor` parameter, which specifies if the calculation rounds up or down, and `transferSharesFrom()` now specifies `floor = false` to round up.

[M-2] Fluctuations in share price affect how much `withdrawLimit` is increased by in `LiquidStakingToken._sync()`

Context:

- LiquidStakingToken.sol#L354-L368
- LiquidStakingToken.sol#L848-L853

Description: When depositing on L2, `_mint()` is called to mint shares to the user:

```
uint256 shares = _totalShares == 0 ? _amount : convertToShares(_amount);
uint256 depositFee = $.syncDepositFee;

// ...

_mintShares(_to, shares);
```

Note that `_amount` is the amount of assets the user should receive, therefore it has to be converted to shares with `convertToShares()`.

Afterwards, when `sync()` is called to sync the L2 deposits to L1, the `RateLimiter` contract is updated with `convertToShares(_amountOut)`:

```
IRateLimiter(getRateLimiter()).updateRateLimit(
    address(this),
    Constants.ETH_ADDRESS,
    convertToShares(_amountOut),
    0
);
```

Note that `_amountOut` here is the sum of unsynced assets minted on L2 through `_mint()`. This is meant to add the amount of shares minted to the user in `_mint()` to the `withdrawLimit`.

However, if `_totalAssets()` increases before `sync()` is called, the share price would increase, causing `convertToShares(_amountOut)` to become less shares than the amount that was minted to the user.

For example:

- Assume that:
  - `totalShares = 100e18`
  - `totalStaked = 100e18`, thus `_totalAssets() = 100e18`
  - `lastAssetsPerShare = 1e18`
- User calls `mint()` to deposit 10 ETH on L2. Assuming that `_amount = 10e18` in `_mint()`:
  - `convertToShares(_amount) = 10e18 * 100e18 / 100e18 = 10e18`

  - The user receives `10e18` shares.
  - Both `totalShares` and `totalStaked` are increased to `110e18`.
- `LiquidStakingTokenLockbox.rebase()` is called on L1:
  - Assume that `_assetsPerShare = 1.5e18` in `_lzReceive()` on L2
  - When `_updateTotalStaked()` is called, `totalStaked = 110e18 * 1.5e18 / 1e18 = 165e18`
- User calls `sync()` to sync L2 deposits to L1:
  - `convertToShares(_amountOut) = 10e18 * 110e18 / 165e18 = ~6.67e18`
  - Only `~6.67e18` shares are added to `withdrawLimit`

As seen from above, `10e18` shares were minted to the user, but only `~6.67e18` shares were added to `withdrawLimit` in the `RateLimiter` contract. This causes users to be unable to withdraw their shares as `withdrawLimit` will be smaller than the actual amount of shares minted.

Recommendation: In the `L2TokenStorage` struct, add a state variable that tracks the amount of unsynced shares minted:

```
uint256 unsyncedShares;
```

In `_mint()`, add the amount of shares minted to `unsyncedShares`:

```

```

In `_sync()`, pass `unsyncedShares` to `updateRateLimit()` instead of `convertToShares(_amountOut)`:

```
  IRateLimiter(getRateLimiter()).updateRateLimit(
      address(this),
      Constants.ETH_ADDRESS,
-     convertToShares(_amountOut),
+     $.unsyncedShares,
      0
  );
+ $.unsynchedShares = 0;
```

This ensures that the exact amount of shares minted in `_mint()` will be added to `withdrawLimit`, and is not susceptible to fluctuations in share price.

Redacted: Fixed in commit 7a84099.

Renascence: Verified, the recommended fix was implemented.

Low Risk

[L-1] `L1SyncPool.withdraw()` also transfers unbacked pxEth out

Context: L1SyncPool.sol#L321-L325

Description: When the owner calls `L1SyncPool.withdraw()`, it transfers out the entire pxEth balance in the `L1SyncPool` contract:

```
uint256 balance = pxEth.balanceOf(address(this));

if (balance == 0) revert Errors.InvalidAmount();

pxEth.transfer(receiver, balance);
```

This means that if the contract holds any unbacked pxEth belonging to unfinalized L2 deposits, they will also be transferred out when this function is called. Should this occur, L2 deposits will no longer be finalizable as the contract will have a deficit of pxEth.

However, this function is only meant to transfer out excess pxEth when the amount received in `_finalizeDeposit()` is more than initially minted in `_anticipatedDeposit()`. As such, if the contract holds any unbacked pxEth, the owner cannot call `withdraw()` to avoid transferred out any unbacked pxEth.

Recommendation: Consider allowing the owner to specify the amount of pxEth to transfer out:

```
function withdraw(address receiver, uint256 amount) external onlyOwner {
    IDineroERC20 pxEth = IDineroERC20(address(getTokenOut()));

    pxEth.transfer(receiver, amount);

    emit Withdraw(receiver, amount);
}
```

Redacted: Fixed in commit f4e5ec9.

Renascence: Verified, the recommended fix was implemented.

[L-2] Inefficient queue implementation for `syncedIds` in `LiquidStakingTokenLockbox`

Context:

- LiquidStakingTokenLockbox.sol#L529
- LiquidStakingTokenLockbox.sol#L825-L833
- LiquidStakingTokenLockbox.sol#L846-L855

Description: In `LiquidStakingTokenLockbox`, the `syncedIds` array is used to implement a queue of finalized L2 deposits. Whenever an L2 deposit is finalized in `_handleFinalizeDeposit()`, its `_syncId` is pushed to the back of the array:

```
$.syncedIds.push(_syncId);
```

In `_syncedIdsBatch()`, the IDs are used sequentially from the front of the array:

```
uint256 size = syncedIdsLen > maxBatch ? maxBatch : syncedIdsLen;

bytes32[] memory syncedIdsBatch = new bytes32[](size);

for (uint256 i = 0; i < size; i++) {
    syncedIdsBatch[i] = $.syncedIds[i];
}

return syncedIdsBatch;
```

Afterwards, in `_updateSyncedIds()`, the remaining unused IDs are copied to the front of the array, and the used IDs are popped:

```
uint256 removeN = maxBatch > syncedIdsLen ? syncedIdsLen : maxBatch;

uint256 newArrayLen = syncedIdsLen - removeN;
for (uint256 i = 0; i < newArrayLen; i++) {
    $.syncedIds[i] = $.syncedIds[i + removeN];
}

for (uint256 i = syncedIdsLen; i > newArrayLen; i--) {
    $.syncedIds.pop();
}
```

However, this implementation is extremely inefficient. Whenever `_updateSyncedIds()` is called, the function will iterate over the entire `syncedIds` array, which costs a huge amount of gas. Furthermore, if the `syncedIds` array happens to be too large, the function could consume too much gas and revert due to an out-of-gas error.

Recommendation: A better implementation would be to store an index, which represents the front of the queue:

```
  bytes32[] syncedIds;
+ uint256 firstSyncedIdIndex;
```

In `_syncedIdsBatch()`, the for-loop should iterate from this index onwards:

```
- for (uint256 i = 0; i < size; i++) {
+ for (uint256 i = $.firstSyncedIdIndex; i < $.firstSyncedIdIndex + size; i++) {
        syncedIdsBatch[i] = $.syncedIds[i];
  }
+ $.firstSyncedIdIndex += size;
```

With this implementation, `_updateSyncedIds()` can be removed as IDs no longer need to be removed from array.

Redacted: Fixed in commit 37be83e.

Renascence: Verified, the recommended fix was implemented.

[L-3] `lastCompletedSyncIndex` is not updated in `LiquidStakingToken._withdrawPendingDeposit()`

Context: LiquidStakingToken.sol#L720-L731

Description: `LiquidStakingToken._withdrawPendingDeposit()` iterates through the `syncIndexPendingAmount` array from `lastCompletedSyncIndex` to `lastPendingSyncIndex` to sequentially subtract `_withdrawAmount` from pending sync batches:

```
for (uint256 i = lastCompletedIndex + 1; i <= lastPendingIndex; i++) {
    uint256 pendingAmount = $.syncIndexPendingAmount[i];

    if (pendingAmount > remaining) {
        $.syncIndexPendingAmount[i] -= remaining;
        remaining = 0;
        break;
    }

    remaining -= pendingAmount;
    $.syncIndexPendingAmount[i] = 0;
}
```

If a sync batch has no remaining pending amount, `syncIndexPendingAmount[i]` will be `0`.

However, `lastCompletedSyncIndex` is not increased during/after the loop even if a sync batch has no remaining amount and is complete. Subsequently, whenever `_withdrawPendingDeposit()` is called, it will continue to loop over the same sync batches that have no remaining amount, which is extremely gas inefficient.

Additionally, since `lastPendingSyncIndex` is unbounded (ie. depends on how many pending sync batches exist), if many pending sync batches exist, it is theoretically possible for the loop to consume too much gas and revert due to an out-of-gas error.

Recommendation: Increment `lastCompletedIndex` at the end of the for-loop:

```
    for (uint256 i = lastCompletedIndex + 1; i <= lastPendingIndex; i++) {
        // ...

        remaining -= pendingAmount;
        $.syncIndexPendingAmount[i] = 0;
+       $.lastCompletedIndex++;
    }
```

This ensures that `lastCompletedIndex` is incremented whenever a sync batch is complete (ie. has no remaining amount).

Redacted: Fixed in commit a3b28f8.

Renascence: Verified, the recommended fix was implemented.

[L-4] Missing `nonReentrant` modifier on user-facing functions

Context:

- LiquidStakingToken.sol
- LiquidStakingTokenLockbox.sol

Description: The `LiquidStakingToken` and `LiquidStakingTokenLockbox` contracts inherit `ReentrancyGuardUpgradeable` to use the `nonReentrant` modifier to prevent reentrancy. However, several functions that can be called by regular users are missing the `nonReentrant` modiifer.

In `LiquidStakingToken`:

- `_lzReceive()` - A user can manually trigger this by calling `EndpointV2.lzReceive()`.
- `deposit()`
- `_sync()` - To prevent users from calling `L2SyncPool.sync()`.

In `LiquidStakingTokenLockbox`:

- `_lzReceive()` - A user can manually trigger this by calling `EndpointV2.lzReceive()`.
- `_handleFinalizeDeposit()` - A user can manually trigger this by calling `OptimismPortal.finalizeWithdrawalTransaction()`.

Currently in the `LiquidStakingToken` contract, only the `withdraw()` function has the `nonReentrant` modifier which makes the modifier redundant.

Recommendation: Add the `nonReentrant` modifier to all functions listed above.

Redacted: Fixed in commit fb65799.

Renascence: Verified, the recommended fix was implemented.