Dinero (Super ETH) Audit Report

Version 2.0

Audited by:

MiloTruck

bytes032

15 November 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 (Super ETH)
Repository	dinero-pirex-eth
Commit Hash	21b2a7a570eb...
Mitigation Hash	163bfa3b4518...
Date	1 November 2024 - 15 November 2024

2.3 Issues Found

Severity	**Count**
High Risk	1
Medium Risk	3
Low Risk	0
Informational	0
Total Issues	**4**

3 Findings Summary

ID	Description	Status
H-1	Incorrect encoding of `wlstAmount` when sending messages to `DineroOFT`	Resolved
M-1	Missing `whenNotPaused` and `nonReentrant` modifiers on `L2SyncPool.depositAndWrap()`	Resolved
M-2	Rounding in `LiquidStakingToken._lzReceive()` could cause wrapping LST tokens to revert	Resolved
M-3	`StargateAdapter.sendMessage()` is incompatible with ETH	Resolved

4 Findings

High Risk

[H-1] Incorrect encoding of `wlstAmount` when sending messages to `DineroOFT`

Context:

- OFTLockbox.sol#L154-L167
- OFTCore.sol#L235-L236
- OFTMsgCodec.sol#L48-L55

Description: In `OFTLockbox.lzCompose()`, when sending wLST tokens to another chain, the `message` passed to `_lzSend()` is encoded as `address receiver || uint256 wlstAmount`:

```
uint256 wlstAmount = IWrappedLiquidStakedToken(address(innerToken))
    .wrap(lstAmount);

if (endpoint.eid() == dstEid) {
    innerToken.safeTransfer(receiver, wlstAmount);
} else {
    _lzSend(
        dstEid,
        abi.encodePacked(receiver, wlstAmount),
        _getOAppOptionsType3Storage().enforcedOptions[dstEid][SEND],
        MessagingFee(msg.value, 0),
        receiver
    );
}
```

However, when the message is received on the destination chain in `OFTCore._lzReceive()`, the amount of OFT tokens to mint (ie. `wlstAmount`) is encoded as shared decimals and decoded into local decimals:

```
// @dev Credit the amountLD to the recipient and return the ACTUAL amount the
recipient received in local decimals
uint256 amountReceivedLD = _credit(toAddress, _toLD(_message.amountSD()),
_origin.srcEid);
```

`OFTMsgCodec.amountSD()` shows that the amount is encoded as a `uint64`:

```
uint8 private constant SEND_TO_OFFSET = 32;
uint8 private constant SEND_AMOUNT_SD_OFFSET = 40;

function amountSD(bytes calldata _msg) internal pure returns (uint64) {
    return uint64(bytes8(_msg[SEND_TO_OFFSET:SEND_AMOUNT_SD_OFFSET]));
}
```

As such, since `wlstAmount` is wrongly encoded as a `uint256` with local decimals, an incorrect amount of `DineroOFT` tokens will be minted on the destination chain. It should be encoded as a `uint64` with

shared decimals instead.

Recommendation: In `OFTLockbox.lzCompose()`, convert `wlstAmount` to shared decimals before encoding it in the message:

```
    } else {
        _lzSend(
            dstEid,
-           abi.encodePacked(receiver, wlstAmount),
+           abi.encodePacked(receiver, _toSD(wlstAmount)),
            _getOAppOptionsType3Storage().enforcedOptions[dstEid][SEND],
            MessagingFee(msg.value, 0),
            receiver
        );
    }
```

`_toSD()` is a function used to convert local decimals into shared decimals, similar to the one in `OFTCore`:

```
function _toSD(uint256 _amountLD) internal view virtual returns (uint64 amountSD) {
    return uint64(_amountLD / decimalConversionRate);
}
```

Note that `decimalConversionRate` should be the same value as `decimalConversionRate` in all `DineroOFT` contracts on all destination chains.

Redacted: Fixed in commit e9a6ad0.

Renascence: Verified, the recommended fix was implemented.

Medium Risk

[M-1] Missing `whenNotPaused` and `nonReentrant` modifiers on `L2SyncPool.depositAndWrap()`

Context: L2SyncPool.sol#L199-L207

Description: In the `L2SyncPool` contract, a new `depositAndWrap()` function was added to allow callers to deposit tokens and wrap them into wLST. However, unlike the `deposit()` function, this function is not overridden with the `nonReentrant` and `whenNotPaused` modifiers in the `LiquidStakingToken` contract, which inherits `L2SyncPool`.

As a result, whenever the `LiquidStakingToken` contract is paused, users can still deposit on L2 using `depositAndWrap()`. Similarly, `depositAndWrap()` can be used for reentrancy attacks.

Recommendation: In `LiquidStakingToken`, add a function that overrides `depositAndWrap()` with both modifiers:

```
function depositAndWrap(
    address tokenIn,
    uint256 amountIn,
    uint256 minAmountOut
)
    public
    payable
    virtual
    override
    nonReentrant
    whenNotPaused
    returns (uint256)
{
    return super.depositAndWrap(tokenIn, amountIn, minAmountOut);
}
```

Redacted: Fixed in commit 1566925.

Renascence: Verified, the recommended fix was implemented.

[M-2] Rounding in `LiquidStakingToken._lzReceive()` could cause wrapping LST tokens to revert

Context:

- LiquidStakingToken.sol#L295-L303
- DineroERC20RebaseUpgradeable.sol#L161-L177

Description: When `_messageType` is specified as `MESSAGE_TYPE_DEPOSIT_WRAP` in `LiquidStakingToken._lzReceive()`, LST tokens are minted and converted to wLST as such:

```
_mintShares(address(this), shares);
$.totalStaked += _amount;

uint256 amount = convertToAssets(shares, false);
_approve(address(this), $.wLST, amount, false);
uint256 wAmount = IWrappedLiquidStakedToken($.wLST).wrap(
    amount
);
IWrappedLiquidStakedToken($.wLST).transfer(_receiver, wAmount);
```

As seen from above, the `amount` of LST tokens to wrap into wLST is calculated with `convertToAssets(shares, false)`, which rounds up. Subsequently, when `WrappedLiquidStakingToken.transfer()` attempts to transfer `amount` of LST tokens, the number of shares to transfer is rounded down:

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

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

    _transferShares(_sender, _recipient, sharesToTransfer);
    _emitTransferEvents(_sender, _recipient, _amount, sharesToTransfer);
}
```

Therefore, the share calculations can be summarized as such:

```
uint256 amount = shares.mulDivUp(_totalAssets(), totalShares);
uint256 sharesToTransfer = amount.mulDivDown(totalShares, _totalAssets());
```

In order for `LiquidStakingToken._lzReceive()` to not revert, `sharesToTransfer` has to be less than `shares`. Otherwise, `WrappedLiquidStakedToken.transfer()` would attempt to transfer more LST shares than the number minted.

However, since the calculation of `amount` rounds up, it is possible for `shares` to be greater than `sharesToTransfer`. This would cause `LiquidStakingToken._lzReceive()` to revert when called, blocking L1 → L2 deposits.

Recommendation: The amount of LST tokens to wrap should round up:

```
- uint256 amount = convertToAssets(shares, false);
+ uint256 amount = convertToAssets(shares, true);
  _approve(address(this), $.wLST, amount, false);
  uint256 wAmount = IWrappedLiquidStakedToken($.wLST).wrap(
      amount
  );
  IWrappedLiquidStakedToken($.wLST).transfer(_receiver, wAmount);
```

Redacted: Fixed in commit ccbfb82.

Renascence: Verified, the recommended fix was implemented.

[M-3] `StargateAdapter.sendMessage()` is incompatible with ETH

Context:

- OFTMinter.sol#L57-L79
- StargateAdapter.sol#L208
- StargateAdapter.sol#L50-L55

Description: `OFTMinter.deposit()` allows users to specify `_tokenIn` as `Constants.ETH_ADDRESS` (ie. `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`) to deposit ETH. `_tokenIn` is then passed to `StargateAdapter.sendMessage()` as `_l2token`:

```
if (_tokenIn != Constants.ETH_ADDRESS) {
    IERC20(_tokenIn).safeTransferFrom(
        msg.sender,
        address(adapter),
        _amountIn
    );
}

// ...

adapter.sendMessage{value: msg.value}(
    lockbox,
    _tokenIn,
    msg.sender,
    data,
    0
);
```

`StargateAdapter.sendMessage()` checks that `_l2token` is whitelisted by the protocol admin in the `tokens` mapping:

```
if (!tokens[_l2token]) revert Errors.UnauthorizedToken();
```

However, whenever the protocol admin calls `whitelistToken()` to whitelist a token, `forceApprove()` is called on the `_token` address:

```
function whitelistToken(address _token) external onlyOwner {
    tokens[_token] = true;
    IERC20(_token).forceApprove(address(stargate), type(uint256).max);

    emit Token(_token, true);
}
```

Therefore, it is not possible to whitelist `Constants.ETH_ADDRESS` in the `tokens` mapping as calling

`forceApprove()` on `Constants.ETH_ADDRESS` will always revert. As a result, depositing ETH through `OFTMinter` is not possible.

Recommendation: Modify `whitelistToken()` and `removeToken()` to only call `forceApprove()` when `_token` is not `Constants.ETH_ADDRESS`:

```
function whitelistToken(address _token) external onlyOwner {
    tokens[_token] = true;
    if (_token != Constants.ETH_ADDRESS) {
        IERC20(_token).forceApprove(address(stargate), type(uint256).max);
    }

    emit Token(_token, true);
}

function removeToken(address _token) external onlyOwner {
    tokens[_token] = false;
    if (_token != Constants.ETH_ADDRESS) {
        IERC20(_token).forceApprove(address(stargate), 1);
    }

    emit Token(_token, false);
}
```

Redacted: Fixed in commit 8e711ad.

Renascence: Verified, the recommended fix was implemented.