RENASCENCE

Dinero (Stargate Bridge) Audit Report

Version 1.0

Audited by:

MiloTruck

bytes032

September 23, 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 (Stargate Bridge)
Repository	dinero-pirex-eth
Commit Hash	6f4fbed86674...
Date	18 September 2024 - 23 September 2024

2.3 Issues Found

Severity	Count
High Risk	2
Medium Risk	4
Low Risk	1
Informational	1
Total Issues	**8**

3 Findings Summary

ID	Description	Status
H-1	Missing `extraOptions` in `StargateAdapter.prepareTransaction()` for `lzCompose()`	Open
H-2	Setting `minAmountLD` to `amountReceivedLD` causes L1 sync pool to receive less ETH than intended	Open
M-1	`LiquidStakingTokenNonNative._sync()` does not handle ETH deposits on L2	Open
M-2	Unsafe ERC-20 operations in `StargateAdapter`	Open
M-3	Using `transfer()` to transfer ETH is unsafe	Open
M-4	Stargate's composability feature cannot be used in bus mode	Open
L-1	Refunds are not sent to the user in `StargateAdapter`	Open
I-1	`_l2TokenIn` address is wrongly sent to L1 instead of `_l1TokenIn`	Open

4 Findings

High Risk

[H-1] Missing `extraOptions` in `StargateAdapter.prepareTransaction()` for `lzCompose()`

Context: StargateAdapter.sol#L151-L159

Description: In `StargateAdapter.prepareTransaction()`, `extraOptions` is set to empty bytes in the parameters sent to `sendToken()`:

```
sendParam = SendParam({
    dstEid: DST_EID,
    to: addressToBytes32(_receiver),
    amountLD: _amountLD,
    minAmountLD: _minAmountLD,
    extraOptions: new bytes(0),
    composeMsg: composeMsg,
    oftCmd: takeBus ? new bytes(1) : new bytes(0)
});
```

`L1StargateReceiverETH` uses `lzCompose` to receive data and an external call from L2. According to Stargate's documentation, to use the `lzCompose` feature, `extraOptions` must specify the amount of gas that `lzCompose()` is called with on L1:

> You also need to pass additional gas for the compose call. You need to set this value to the amount of gas your `lzCompose` function in the compose receiver consumes.

However, since `extraOptions` is set to empty bytes in the current implementation, the L2 → L1 call to `L1StargateReceiverETH` will not work.

Recommendation: Add the minimum gas limit that `lzCompose()` is called with on L1 to `extraOptions`. This can be done by following the following documentation:

- Stargate's documentation
- LayerZero's documentation

[H-2] Setting `minAmountLD` to `amountReceivedLD` causes L1 sync pool to receive less ETH than intended

Context:

- StargateAdapter.sol#L205-L209
- StargateAdapter.sol#L151-L162
- L1SyncPool.sol#L276

Description: In `StargateAdapter.prepareTransaction()`, `amountLD` is first set to `amountIn` and `minAmountLD` is first set to `amountOut`:

```
(
    uint256 valueToSend,
    SendParam memory sendParam,
    MessagingFee memory messagingFee
) = prepareTransaction(amountIn, amountOut, _receiver, _message);
```

```
sendParam = SendParam({
    dstEid: DST_EID,
    to: addressToBytes32(_receiver),
    amountLD: _amountLD,
    minAmountLD: _minAmountLD,
    extraOptions: new bytes(0),
    composeMsg: composeMsg,
    oftCmd: takeBus ? new bytes(1) : new bytes(0)
});
```

Afterwards, `minAmountLD` is overwritten with `amountReceivedLD` returned from `quoteOFT()`:

```
(, , OFTReceipt memory receipt) = stargate.quoteOFT(sendParam);
sendParam.minAmountLD = receipt.amountReceivedLD;
```

`amountReceivedLD` is the amount of tokens that will be sent to L1 after Stargate's transfer limits are applied and fees are subtracted.

However, since `minAmountLD` is the minimum amount of tokens that L1 should receive, setting it to `amountReceivedLD` is dangerous as there is effectively no lower bound on the amount of tokens received. For example, if Stargate's transfer limits are reached and `quoteOFT()` returns `amountReceivedLD = 0`, it becomes possible for `stargate.sendToken()` to send no tokens to L1 when `sync()` is called.

This breaks the syncing of L2 deposits as L1 sync pool always expects to receive `amountIn` of ETH from slow sync:

```
if (amountIn != msg.value) revert Errors.InvalidAmount();
```

Recommendation: In `sendMessage()`, specify `_minAmountLD` as `amountIn` instead of `amountOut`:

```
  (
      uint256 valueToSend,
      SendParam memory sendParam,
      MessagingFee memory messagingFee
- ) = prepareTransaction(amountIn, amountOut, _receiver, _message);
+ ) = prepareTransaction(amountIn, amountIn, _receiver, _message);
```

In `prepareTransaction()`, remove the lines that overwrite `minAmountLD` with `amountReceivedLD`:

```
- (, , OFTReceipt memory receipt) = stargate.quoteOFT(sendParam);
- sendParam.minAmountLD = receipt.amountReceivedLD;
```

This ensures that L1 will always receive `amountIn` of ETH from slow sync.

Additionally, if Stargate protocol fees are enabled for the pool being used, `amountLD` must include the ETH used for Stargate fees (ie. `amountLD = amountIn + stargateFees`). This means `msg.value` sent by the user will have to be equal to `amountIn + stargateFee + messagingFee.nativeFee`.

Medium Risk

[M-1] `LiquidStakingTokenNonNative._sync()` **does not handle ETH deposits on L2**

Context: LiquidStakingTokenNonNative.sol#L86-L93

Description: `LiquidStakingTokenNonNative._sync()` transfers `_l2TokenIn` to `StargateAdapter`, followed by `msg.value - nativeFee` of ETH when calling `sendMessage()`:

```
//transfer funds to messenger
IERC20(_l2TokenIn).safeTransfer(getMessenger(), _amountIn);

// send slow sync message
uint256 messageServiceFee = msg.value - _fee.nativeFee;
ICrossDomainMessenger(getMessenger()).sendMessage{
    value: messageServiceFee
}(getReceiver(), data, _minGasLimit());
```

However, this implementation only works when `_l2TokenIn` is an ERC-20 token. If `_l2TokenIn` is `Constants.ETH_ADDRESS`, the call to `safeTransfer()` would revert and `amountIn` of ETH would not be transferred to `StargateAdapter`.

As such, if `LiquidStakingTokenNonNative` is used to sync L2 deposits, it would revert when attempting to handle ETH deposits on L2.

Additionally, note that Stargate V2 does not support cross-asset bridging (ie. to receive ETH on L1, ETH must be sent from L2). This means that `LiquidStakingTokenNonNative` can only be used with ETH deposits, since L1 sync pool expects slow sync to send ETH from L2 to L1.

Recommendation: Refactor `LiquidStakingTokenNonNative._sync()` to handle ETH deposits on L2:

```
-  //transfer funds to messenger
-  IERC20(_l2TokenIn).safeTransfer(getMessenger(), _amountIn);

+ if (_l2TokenIn != Constants.ETH_ADDRESS) {
+      //transfer funds to messenger
+      IERC20(_l2TokenIn).safeTransfer(getMessenger(), _amountIn);
+ }

  // send slow sync message
  uint256 messageServiceFee = msg.value - _fee.nativeFee;
+ if (_l2TokenIn == Constants.ETH_ADDRESS) messageServiceFee += _amountIn;
  ICrossDomainMessenger(getMessenger()).sendMessage{
      value: messageServiceFee
  }(getReceiver(), data, _minGasLimit());
```

[M-2] Unsafe ERC-20 operations in `StargateAdapter`

Context:

- StargateAdapter.sol#L54
- StargateAdapter.sol#L65
- StargateAdapter.sol#L99

Description: `StargateAdapter` directly calls ERC20's `approve()` and `transfer()` functions, which might not work for tokens that are non-compliant with the ERC20 standard. The instances of unsafe ERC20 operations are:

- In `whitelistToken()`:

```
IERC20(_token).approve(address(stargate), type(uint256).max);
```

- In `removeToken()`:

```
IERC20(_token).approve(address(stargate), 0);
```

- In `withdraw()`:

```
IERC20(_token).transfer(msg.sender, _amount);
```

Additionally, if `removeToken()` was called with BNB, `approve()` would revert as BNB reverts on zero-value approvals:

```
/* Allow another contract to spend some tokens in your behalf */
function approve(address _spender, uint256 _value)
    returns (bool success) {
    if (_value <= 0) throw;
    allowance[msg.sender][_spender] = _value;
    return true;
}
```

Therefore, `removeToken()` can never be called with BNB.

Recommendation: Use OpenZeppelin's `SafeERC20` library to handle these operations. `approve()` can be replaced with `forceApprove()`, while `transfer()` should be replaced with `safeTransfer()`.

If there is ever a need to transfer BNB in `StargateAdapter`, consider resetting the allowance to 1 wei in `removeToken()`:

```
- IERC20(_token).approve(address(stargate), 0);
+ IERC20(_token).approve(address(stargate), 1);
```

[M-3] Using `transfer()` to transfer ETH is unsafe

Context:

- StargateAdapter.sol#L97

Description: `StargateAdapter.withdraw()` uses `.transfer()` to transfer ETH to the owner:

```
payable(msg.sender).transfer(_amount);
```

However, `.transfer()` does not work on certain chains (eg. ZKSync) due to the fixed 2300 gas stipend. Moreover, its use is discouraged on other chains as gas costs can change in the future.

See these resources:

- https://twitter.com/zksync/status/1644139364270878720
- https://consensys.io/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Recommendation: Use a low-level call to transfer ETH instead:

```
- payable(msg.sender).transfer(_amount);
+ payable(msg.sender).call{value: _amount}("");
```

[M-4] Stargates composability feature cannot be used in bus mode

Context:

- StargateAdapter.sol#L105-L115
- L1StargateReceiverETH.sol#L75-L107

Description: The owner of `StargateAdapter` can configure the `takeBus` state variable, which determines if L2 → L1 bridging is performed through Stargate's taxi or bus mode:

```
/**
* @notice Set if the bus ride should be taken
*  - Taxi: Immediately sends an omnichain message to the destination chain.
*  - Bus: Transaction batching. The message will be sent to destination chain when a
"bus" reaches a set number of passengers
* @param _takeBus bool to set if the bus ride should be taken
*/
function setTakeBusRide(bool _takeBus) external onlyOwner {
    takeBus = _takeBus;

    emit TakeBusRide(_takeBus);
}
```

On L1, the protocol uses `lzCompose()` to receive L2 messages and perform actions when messages are received.

However, according to Stargate's documentation, the `lzCompose()` feature can only be used in taxi mode:

> Note: Only Stargate's `taxi()` method is composable, you cannot perform destination logic with `rideBus()`.

Therefore, if `takeBus` is enabled, `lzCompose()` will not be called on L1, causing L2 deposits to never be synced.

Recommendation: Remove the option to use bus mode from `StargateAdapter` and only use taxi mode.

Low Risk

[L-1] Refunds are not sent to the user in `StargateAdapter`

Context:

- StargateAdapter.sol#L213-L215
- LiquidStakingTokenNonNative.sol#L62-L74

Description: When sending a fast sync message directly through LayerZero, the refund address is specified as the user:

```
// send fast sync message
MessagingReceipt memory receipt = _lzSend(
    // ...
    MessagingFee(_fee.nativeFee, 0),
    payable(msg.sender)
);
```

In contrast, in `StargateAdapter.sendMessage()`, the refund address is set to `address(this)` when calling `stargate.sendToken()`:

```
(msgReceipt, oftReceipt, ticket) = stargate.sendToken{
    value: valueToSend
}(sendParam, messagingFee, address(this));
```

This is inconsistent and might cause the user calling `sync()` to lose funds, since they only receive a portion of refunds. A user might call `sync()` with excess native fees for LayerZero/Stargate messaging and expect all excess ETH to be refunded to them, but instead, excess ETH for Stargate messaging is left in the `StargateAdapter` contract.

Recommendation: Pass caller's address to `StargateAdapter.sendMessage()` from `LiquidStakingTokenNonNative._sync()`, and specify the caller as the refund address when calling `sendToken()` instead.

Informational

[I-1] `_l2TokenIn` address is wrongly sent to L1 instead of `_l1TokenIn`

Context:

- LiquidStakingToken.sol#L851-L857
- LiquidStakingTokenNonNative.sol#L78-L84

Description: When building the message for slow sync in `LiquidStakingToken()`, `_l1TokenIn` is passed to L1:

```
bytes memory data = abi.encode(
    endpoint.eid(),
    receipt.guid,
    _l1TokenIn,
    _amountIn,
    _amountOut
);
```

In contrast, the message in `LiquidStakingTokenNonNative` sends the `_l2TokenIn` address to L1 instead:

```
bytes memory data = abi.encode(
    endpoint.eid(),
    receipt.guid,
    _l2TokenIn,
    _amountIn,
    _amountOut
);
```

This is inconsistent. If `_l1TokenIn` and `_l2TokenIn` are different addresses, this could result in the wrong token address being passed to L1.

Recommendation: In `LiquidStakingTokenNonNative._sync()`, send the `_l1TokenIn` address instead:

```
  bytes memory data = abi.encode(
      endpoint.eid(),
      receipt.guid,
-     _l2TokenIn,
+     _l1TokenIn,
      _amountIn,
      _amountOut
  );
```