CANTINA

Decent Oft4626

Competition

October 9, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Decent abstracts all of the complexities associated with various chains and tokens, so user activity can seamlessly scale with the number of chains. Decent meets users where they are to create a unified experience across chains.

From Sep 19th to Sep 23rd Cantina hosted a competition based on Decent-oft4626. The participants identified a total of **17** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 7
- Gas Optimizations: 0
- Informational: 10

The present report only outlines the **low** risk issues.

3 Findings

3.1 Low Risk

3.1.1 `QuoteRebase(...)` calculates incorrrect fee because `_newExchangeRate` is encoded twice

Submitted by GeneralKay

Severity: Low Risk

Context: L1Oft4626Router.sol#L158

Description: The `quoteRebase(...)` function is used to estimate the gas fee for the `rebase(...)` function. However there is an issue in the `quoteRebase(...)` function and the issue is that the `_newExchangeRate` is encoded twice to get the `_payload` instead of just once.

Proof of Concept: The main issue is that the `abi.encode(...)` function is already called on the `_newExchangeRate` in the `_generateRebasePayload(...)` function then another second `abi.encode(...)` function is called on the returned value from the first `abi.encode(...)` function in the `_generateRebasePayload(...)` function.

This causes the `_newExchangeRate` to be encoded twice before the quote is determined. This means that the `_payload` is a wrong value because it is the result of encoding `_newExchangeRate` twice instead of once.

Below is the `quoteRebase(...)` function of the `L1Oft4626Router.sol` contract:

- `L1Oft4626Router.sol`:

```
function quoteRebase(uint32 dstEid) external view returns (uint256 nativeFee) {
    uint256 _newExchangeRate = getExchangeRate();
    bytes memory _payload = abi.encode(_generateRebasePayload(_newExchangeRate));
    bytes memory _options = OptionsBuilder.newOptions().addExecutorLzReceiveOption(gasLzReceive, 0);
    MessagingFee memory fee = _quote(dstEid, _payload, _options, false);
    return (fee.nativeFee);
}
```


And here we can see that `_generateRebasePayload(...)` already encoded the new exchange rate.

- `L1Oft4626Router.sol`:

```
function _generateRebasePayload(uint256 _newRate) internal view returns (bytes memory _payload) {
        _payload = abi.encode(_newRate, block.timestamp);
}
```

Below we can see how the main `rebase(...)` function only calls the `_generateRebasePayload(...)` function to encode the exchange rate without a second `abi.encode` on the return value like the `quoteRebase(...)` function.

- `L1Oft4626Router.sol`:

```
function rebase(uint32 dstEid) external payable {
    uint256 _newExchangeRate = getExchangeRate();
    oft4626.setExchangeRate(_newExchangeRate);
    bytes memory _payload = _generateRebasePayload(_newExchangeRate);
    bytes memory _options = OptionsBuilder.newOptions().addExecutorLzReceiveOption(gasLzReceive, 0);
    _lzSend(
        dstEid,
        _payload, // (uint256,uint256) exchangeRate, timestamp
        _options, // Message execution options (e.g., gas to use on destination).
        MessagingFee(msg.value, 0), // Fee struct containing native gas and ZRO token.
        payable(msg.sender) // The refund address in case the send call reverts.
    );
}
```

Recommendation: Consider removing the second encoding since the `_generateRebasePayload(...)` function already encoded the `_newExchangeRate`.

```
  function quoteRebase(uint32 dstEid) external view returns (uint256 nativeFee) {
      uint256 _newExchangeRate = getExchangeRate();
-     bytes memory _payload = abi.encode(_generateRebasePayload(_newExchangeRate));
+     bytes memory _payload = _generateRebasePayload(_newExchangeRate);
      bytes memory _options = OptionsBuilder.newOptions().addExecutorLzReceiveOption(gasLzReceive, 0);
      MessagingFee memory fee = _quote(dstEid, _payload, _options, false);
      return (fee.nativeFee);
  }
```

3.1.2 Stale exchange rate could lead to inaccurate asset-to-share conversions

Submitted by 0x4non

Severity: Low Risk

Context: OftERC4626.sol#L67-L73

Description: The `OftERC4626` contract relies on an `exchangeRate` variable to convert between assets and shares in the `assetsToShares` and `sharesToAssets` functions. However, the `exchangeRate` may become outdated if not updated regularly, as it is set manually via the `_setExchangeRate` function, which updates the `lastUpdate` timestamp. An outdated `exchangeRate` can lead to inaccurate conversions between assets and shares, resulting in unfair valuations for users depositing or withdrawing assets. This discrepancy can cause financial losses or enable arbitrage opportunities. A similar issue occurred with the Venus protocol during the LUNA collapse, where stale price data led to significant losses (Venus and Blizz Finance Exploit).

Recommendation: Implement safeguards to ensure the `exchangeRate` used for conversions is up-to-date:

- Validate Exchange Rate Freshness: Before performing conversions in `assetsToShares` and `sharesToAssets`, check if the `exchangeRate` is recent by verifying that `block.timestamp - lastUpdate` is within an acceptable threshold. If the exchange rate is outdated, revert the transaction to prevent unfair conversions.
- Automate Exchange Rate Updates: Integrate an automatic mechanism to update the `exchangeRate`, such as using a trusted oracle service or scheduling periodic updates via smart contract automation tools (e.g., Chainlink Keepers).

By ensuring the `exchangeRate` remains current, the contract can provide accurate and fair conversions between assets and shares, protecting users from potential losses due to outdated valuations.

3.1.3 Approvals being tracked in assets leads to multiple issues

Submitted by J4X, also found by pep7siup and 0xarno

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `OftERC4626` contract allows users to set approvals for shares. These are then manually converted to assets.

```
function approveShares(address spender, uint256 value) public virtual returns (bool) {
    address owner = _msgSender();
    uint256 _assets = sharesToAssets(value);
    _approve(owner, spender, _assets);
    return true;
}
```

Effectively, we are converting `WstETH=shares` → `stETH=assets`. As `stETH` is rebasing, the supply gradually increases while it is wrapped. However, this design leads to a few problems when users want to use approvals.

1. Frontrun exchange rate update: A user could have received approval for 100 shares, which would be equal to 100 assets. By now, the exchange rate has increased, and someone calls to update it to 1:1.05. The user sees this and quickly transfers the shares of the other user to him before the value increases, but he can't anymore. Effectively he now got more assets than he should.

2. Frontrun `transferFrom()`: The same issue can also occur the other way around. Imagine the rate is currently 1:1, and a user approves a DeFi protocol for the exact amount of shares equal to the assets. Now, meanwhile, the rate increases to 1:1.00001. Before a user is able to trigger the DeFi protocol, someone front-runs his call and updates the exchange rate. As the assets are now a minor bit too few, the whole transfer will not work anymore.

Recommendation: The issue can be mitigated in 2 ways, of which one works much better than the other.

1. Force exchange rate update on any transfer: The first solution would be to enforce an exchange rate update before any transfer. However, this will only work on the L1 side, so it is suboptimal.
2. Track approvals in shares instead of assets: If the approvals are tracked in shares instead of assets, the exchange rate does not influence them. As the actual transfer is also done in shares, this works strictly as intended and does not allow for any of the damage scenarios above.

3.1.4 User receives fewer shares than quoted due to fee change, causing undesirable loss

Submitted by pep7siup

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

If the `feeManager.feeBps` is increased just before a `depositAndSend` transaction is confirmed, users may receive fewer shares than originally quoted through the `quoteDepositAndSend` function. This leads to loss of funds for users.

Proof of Concept: The root cause of the vulnerability is that the fee can be changed by an admin right before the `depositAndSend` operation, impacting the amount of shares received by the user. Let us walk through the issue with the following scenario:

1. Alice calls `quoteDepositAndSend` to get an estimate of the shares she would receive for her deposit.
2. The admin raises the protocol fee (via `feeManager.feeBps`) after Alice receives the quote but before she executes the `depositAndSend`.
3. As a result, Alice receives fewer shares than expected due to the increased fee, which was not accounted for in the original quote, leading to an effective loss of funds.


```
80:     function depositAndSend(DecentBridgeCall memory bridgeCall) public payable {
         // ...
84:         uint256 shares = _depositToVault(bridgeCall.amount);
85: =>          uint256 fee = _feeTaken(shares); // @audit: more fees got deducted if fee increased before
↪  depositAndSend was confirmed
86:         // mint correlating oft shares
         // ...
96:     }
```

Recommended: Add slippage protection to the `depositAndSend` function. For example, the `DecentBridgeCall` struct should incorporate a `minAmountOut` for the expected shares to ensure users receive at least the quoted amount or revert the transaction.

3.1.5 The conversation rate may leads to loss of funds

Submitted by Nyksx

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Layerzero has the concept of local and shared decimals. Local decimals are 18 in the `L1Oft4626Router`, and shared decimals are 6 in the OFTCore. Therefore, the `decimalConversationRate` is equal to 10^{12}. This means that the user can only transfer amounts that are multiples of 10^{12}. If the amount is not multiple of 10^{12}, then the dust amount is chopped from the amount and will be refunded to the user. The `depositAndSend` function takes a fee from the share amount and then the new `bridgeCall.amount` will be equal to `shares - fee`.

```
function depositAndSend(DecentBridgeCall memory bridgeCall) public payable {
    // transfer specified amount of vault asset to contract
    IERC20(vault.asset()).safeTransferFrom(msg.sender, address(this), bridgeCall.amount);
    // deposit amount to vault, Oft4626Router receives shares
    uint256 shares = _depositToVault(bridgeCall.amount);
    uint256 fee = _feeTaken(shares);
    // mint correlating oft shares
    oft4626.mintShares(address(this), shares);
    if (fee > 0) {
        oft4626.transferShares(feeManager.feeRecipient, fee);
    }
    // update bridgeCall amount to shares
    bridgeCall.amount = shares - fee;
    // burn shares + send to dst chain
    uint256 amountSd = _sendOft(bridgeCall);

    _refundDust(bridgeCall.amount - amountSd, bridgeCall.refundAddress);
}
```

The problem is that if a user wants to send 10^{12} tokens, no dust needs to be removed, but when the fee is applied, the `bridgeCall.amount` will be less than 10^{12}, and when this amount is divided by the `decimalConversationRate` (10^{12}) in the `_getCallParams` function, the final amount will equal 0.

Impact: The function does not revert when the final amount is 0, so the user loses the fee amount and gets 0 tokens on the destination chain.

Even if the refund happens at the end of the function, the user still loses the fee.

Proof of Concept: See `ApeETH.t.sol`:

```
function test_depositAndSendWithFee() public {
    _rebase();
    FeeManager memory _feeManager = FeeManager({ feeRecipient: TEST.EOA.alice, feeBps: 100 });
    vm.prank(TEST.L1.apeETHRouter.owner());
    TEST.L1.apeETHRouter.updateFeeManager(_feeManager);

    _deposit(10 ** 12, TEST.EOA.bob, TEST.EOA.refund);
    uint256 sharesOfAfter = TEST.L2.apeETH.sharesOf(TEST.EOA.bob);

    assertEq(sharesOfAfter, 0);
}
```

Recommendation: Revert the transaction if the return amount in the `getAmountSD` function is 0 so the user wont lose the fee amount.

```
}
```

}

```
There is no withdraw event emission being implemented here  as per the ERC4626:

```solidity
/**
 * @dev Burns exactly shares from the owner and sends assets of underlying tokens to the receiver.
 *
 * - MUST emit the Withdraw event.<----------------ISSUE------------------->
 * - MAY support an additional flow in which the underlying tokens are owned by the Vault contract before the
 * Redeem execution, and are accounted for during redemption.
 * - MUST revert if all of the shares cannot be redeemed (due to withdrawal limit being reached, slippage, the
 ↪  owner
 *   not having enough shares, etc).
 *
 * NOTE: some implementations will require pre-requesting to the Vault before a withdrawal may be performed.
 * Those methods should be performed separately.
 */
function redeem(uint256 shares, address receiver, address owner) external returns (uint256 assets);
```

Recommendation: emit a withdraw event as per the erc. -->
```

3.1.6 Missing refund mechanism in `withdrawAndSendShares` function of `L2Oft4626Router` contract

Submitted by 0xarno, also found by pep7siup and GeneralKay

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `L2Oft4626Router` contract contains a issue in the `withdrawAndSendShares` function. Specifically, the function lacks a refund mechanism after executing the `_sendOft` operation. This omission is highlighted by the `@audit` tag in the source code:

```
function withdrawAndSendShares(DecentBridgeCall memory bridgeCall) public payable {
    oft4626.transferSharesFrom(msg.sender, address(this), bridgeCall.amount);
    _sendOft(bridgeCall);
    // @audit missing refund
    //  this >       + uint256 amountSD = _sendOft(bridgeCall);
    // and  >        + _refundDust(bridgeCall.amount - amountSD, bridgeCall.refundAddress);
}
```

Impact:

- Token Loss: Without a refund mechanism, any excess tokens resulting from the `_sendOft` operation remain trapped within the contract. Over time, this can lead to significant token accumulation within the contract, reducing the overall token supply and potentially impacting the token's value.

Recommendation: To address the identified issue, the `L2Oft4626Router` contract should implement a refund mechanism in the `withdrawAndSendShares` function. This ensures that any residual tokens resulting from the `_sendOft` operation are appropriately returned to the user.

- Implement Refund Mechanism: Add the missing refund logic to the `withdrawAndSendShares` function to handle any dust (residual tokens) after the `_sendOft` operation.

```
function withdrawAndSendShares(DecentBridgeCall memory bridgeCall) public payable {
    oft4626.transferSharesFrom(msg.sender, address(this), bridgeCall.amount);
    uint256 amountSD = _sendOft(bridgeCall);
    _refundDust(bridgeCall.amount - amountSD, bridgeCall.refundAddress);
}
```

3.1.7 Each update rate of L2 can be arbitrage

Submitted by yttriumzz

Severity: Low Risk

Context: L2Oft4626Router.sol#L73

Description: The Decent protocol allows anyone to synchronize the exchange rate of L1 to L2. However, every exchange rate update on L2 may be arbitraged by attackers. The attack steps are as follows:

1. The attacker bought oft4626 tokens on L2.
2. The oft4626 exchange rate on L2 is updated.
3. The attacker's balance increases.
4. The attacker sold oft4626 tokens on L2.

The above steps can be performed in the same block.

Recommendation: Implement a linear update exchange rate on L2.