Decent

Scope

The code under review can be found within the C4 Decent repository.

Summary

In this contest, I discovered 3 out of the 4 High-severity issues and 2 out of the 5 Medium-severity issues found.

Findings

ID	Issues	Severity
H-01	Missing onlyOwner modifier on function setRouter() in DcntEth contract	High
H-02	dcntEth is transferred to the destination's DecentBridgeAdapter instead of the user	High
H-03	WETH is transferred to incorrect address in function _executeWeth()	High
M-01	Users can bypass fees by using receiveFromBridge() function directly in UTB.sol	Medium
M-02	Incorrect refund address provided to DecentEthRouter causes ETH to be locked	Medium

Findings

[H-01] Missing onlyOwner modifier on function setRouter() in DcntEth contract

Impact

In the DcntEth.sol contract, the function setRouter() is used to set the address for the decentEthRouter being used. The function does not have the onlyOwner modifier applied to it though. Due to this the following could occur:

1. An attacker can brick calls from the expected decentEthRouter address. For example, blocking users from adding or removing their WETH or ETH liquidity.
2. An attacker can mint or burn unlimited tokens for any account in the DcntEth contract.

Proof of Concept

In the below code, we can see that the function can be called by anyone to set the router.

```
File: DcntEth.sol
    function setRouter(address _router) public {
        router = _router;
    }
```

Due to this, the functions below in the DecentEthRouter contract will not work since the mint() and burn() calls are not accessible to this router once the attacker sets it to his router.

```
File: DecentEthRouter.sol
    /// @inheritdoc IDecentEthRouter
    function addLiquidityEth()
        public
        payable
        onlyEthChain
        userDepositing(msg.value)
    {
        weth.deposit{value: msg.value}();
        dcntEth.mint(address(this), msg.value);
    }

    /// @inheritdoc IDecentEthRouter
    function removeLiquidityEth(
        uint256 amount
    ) public onlyEthChain userIsWithdrawing(amount) {
        dcntEth.burn(address(this), amount);
        weth.withdraw(amount);
        payable(msg.sender).transfer(amount);
    }

    /// @inheritdoc IDecentEthRouter
    function addLiquidityWeth(
        uint256 amount
    ) public payable userDepositing(amount) {
        weth.transferFrom(msg.sender, address(this), amount);
        dcntEth.mint(address(this), amount);
    }

    /// @inheritdoc IDecentEthRouter
    function removeLiquidityWeth(
        uint256 amount
    ) public userIsWithdrawing(amount) {
        dcntEth.burn(address(this), amount);
        weth.transfer(msg.sender, amount);
    }
```

Once the router has been set by the attacker, he gains access to the mint() and burn() functions below since he is now able to bypass the onlyRouter modifier. This can be used to mint or burn any number of dcntEth tokens to/from any address.

```
File: DcntEth.sol
    function mint(address _to, uint256 _amount) public onlyRouter {
        _mint(_to, _amount);
    }

    function burn(address _from, uint256 _amount) public onlyRouter {
        _burn(_from, _amount);
    }
```

Tools Used

Manual Review

Recommended Mitigation Steps

Apply the onlyOwner modifier to the function setRouter().

[H-02] dcntEth is transferred to the destination's DecentBridgeAdapter instead of the user

Impact

In the onOFTReceived() function in the decentEthRouter contract, if the WETH balance of the contract is less than the amount needed to send, the dcntEth tokens are transferred instead.

The issue is that if the call originates from the bridgeAndExecute() function on the source chain, the address that should receive the tokens on the destination chain is encoded as the destination decentBridgeAdapter instead of the user. Due to this, the tokens would be permanently locked in the destination's adapter and the user would lose those tokens.

Proof of Concept

Over here, we can see that the _toAddress is encoded in the payload. This _toAddress is passed in as the destination bridge adapter from the decentBridgeAdapter on the source chain as seen here. See the code snippets below for proof:

See Line 120:

```
File: DecentBridgeAdapter.sol
118:         router.bridgeWithPayload{value: msg.value}(
119:             lzIdLookup[dstChainId],
120:             destinationBridgeAdapter[dstChainId],
121:             swapParams.amountIn,
122:             false,
123:             dstGas,
124:             bridgePayload
125:         );
```

See Line 106:

```
File: DecentEthRouter.sol
100:         if (msgType == MT_ETH_TRANSFER) {
101:             payload = abi.encode(msgType, msg.sender, _toAddress,
deliverEth);
102:         } else {
103:             payload = abi.encode(
104:                 msgType,
105:                 msg.sender,
106:                 _toAddress,
107:                 deliverEth,
108:                 additionalPayload
109:             );
110:         }
```

Due to this, when the payload is decoded on the destination chain as seen below in the onOFTReceived() function, the _to address is set to the destination's decentBridgeAdapter.

```
File: DecentEthRouter.sol
238:     function onOFTReceived(
239:         uint16 _srcChainId,
240:         bytes calldata,
241:         uint64,
242:         bytes32,
243:         uint _amount,
244:         bytes memory _payload
245:     ) external override onlyLzApp {
246:         (uint8 msgType, address _from, address _to, bool deliverEth)
= abi
247:             .decode(_payload, (uint8, address, address, bool));
```

Now, if the WETH balance of the destination's decentEthRouter is less than the _amount of WETH being requested, the code would transfer dcntEth. But this transfer would be done to the destination's decentBridgeAdapter, where the tokens would be permanently locked. Following the incorrect transfer, we halt the execution by returning early.

```
File: DecentEthRouter.sol
        if (weth.balanceOf(address(this)) < _amount) {
            dcntEth.transfer(_to, _amount);
            return;
        }
```

Through this, we can see how the user loses his tokens and funds are accumulated in the destination chain's adapter.

Tools Used

Manual Review

Recommended Mitigation Steps

Consider providing the _toAddress on the source chain as the user who initiated the bridgeAndExecute() transaction.

[H-03] WETH is transferred to incorrect address in function _executeWeth()

Impact

In the function _executeWeth(), if the call to the `target` (i.e. destination bridge adapter) fails (due payload call failure or some other reason) or the target partially spends the WETH tokens, the remaining WETH is transferred to the `from` address. This `from` address is the source chain's decentBridgeAdapter contract. The issue is that the WETH will be transferred to the equivalent of the `from` address (src chain adapter) but on the destination chain, which may or may not be owned by anyone. This would mean loss of WETH for the user who was expecting the tokens.

Proof of Concept

1. If the user calls the bridgeAndExecute() function on the source chain, the `from` address is set to be the msg.sender over here. This msg.sender is the decentBridgeAdapter contract on the source chain since it is the one that calls the bridgeWithPayload() function on the source router (see here). The execution path would like as follows:

bridgeAndExecute (UTB) => callBridge (UTB) => bridge (Adapter) => bridgeWithPayload (Router)

2. This `from` address is then decoded on the destination chain in the function onOFTReceived().

```
File: DecentEthRouter.sol
    function onOFTReceived(
        uint16 _srcChainId,
        bytes calldata,
        uint64,
        bytes32,
243:         uint _amount,
244:         bytes memory _payload
245:     ) external override onlyLzApp {
246:         (uint8 msgType, address _from, address _to, bool deliverEth)
= abi
247:             .decode(_payload, (uint8, address, address, bool));
```

3. The onOFTReceived() function calls the execute() function on the DecentBridgeExecutor, which calls the _executeWeth() function. The following occurs in it:

- On Line 31-33, the target contract (destination adapter) is called with the callPayload (i.e. receiveFromBridge function). If this call fails due to payload call failure after post bridge swap or some other reason, the call would revert causing the value of success of be false.
- On Line 35-36, since the call to target failed, the whole WETH amount is transferred to the `from` address, which is the source chain's adapter address being used by someone or no one on the destination chain. Due to this, the tokens the user expects will be lost forever.
- On Line 37, we return early.
- Note - the same issue exists on Line 44 where if the target contract spends WETH partially, the remaining WETH is transferred to the `from` address incorrectly.

```
File: DecentBridgeExecutor.sol
24:     function _executeWeth(
25:         address from,
26:         address target,
27:         uint256 amount,
28:         bytes memory callPayload
29:     ) private {
30:         uint256 balanceBefore = weth.balanceOf(address(this));
31:         weth.approve(target, amount);
32:
33:         (bool success, ) = target.call(callPayload);
34:
35:         if (!success) {
36:             weth.transfer(from, amount);
37:             return;
38:         }
39:
40:         uint256 remainingAfterCall = amount -
41:             (balanceBefore - weth.balanceOf(address(this)));
42:
43:         // refund the sender with excess WETH
44:         weth.transfer(from, remainingAfterCall);
45:     }
```

Through this, we can see that the incorrect `from` address can cause loss of tokens for the user.

Tools Used

Manual Review

Recommended Mitigation Steps

Consider setting the from address in the payload encoding on the source chain to the user who initiates the bridgeAndExecute() transaction.

[M-01] Users can bypass fees by using receiveFromBridge() function directly in UTB.sol

Impact

In the UTB.sol contract, the receiveFromBridge() function is used to call _swapAndExecute() when it receives a call from another chain. The issue with the function is that it is public without any modifier to restrict calls only from the decent bridge adapter or stargate bridge adapter. Due to this, an attacker or user could avoid paying the fees for same chain swap transactions by calling the function directly.

The bypassing of fees would mean major loss of protocol revenue for the team. Due to this, the issue has been marked as high-severity.

Proof of Concept

Here is the whole process:

[A] Execution path for a normal swapAndExecute() transaction:

swapAndExecute (UTB) => retrieveAndCollectFees (UTB) => collectFees (UTBFeeCollector) => _swapAndExecute (UTB) => performSwap (UTB) => swap (UniSwapper) => swapNoPath/swapExactIn/swapExactOut (UniSwapper) => call continues to withdraw tokens through executor for user

[B] Execution path for bypassing fees

receiveFromBridge() (UTB) => _swapAndExecute (UTB) => performSwap (UTB) => swap (UniSwapper) => swapNoPath/swapExactIn/swapExactOut (UniSwapper) => call continues to withdraw tokens through executor for user

As we can see above, [A] goes through the UTBFeeCollector to collect the fees from the user while [B] completely skips over that part since there is a direct call to the internal function _swapAndExecute().

Tools Used

Manual Review

Recommended Mitigation Steps

Restrict the calls to receiveFromBridge() from only the decent bridge adapter and stargate bridge adapter contracts. If more adapters will be added, consider maintaining a mapping or some kind of verification when a call is made from the adapter to the receiveFromBridge() function in UTB.sol.

[M-02] Incorrect refund address provided to DecentEthRouter causes ETH to be locked

Impact

Excess msg.value provided for gas is permanently locked in the DecentBridgeAdapter contract.

Proof of Concept

1. The below code snippet is from the _bridgeWithPayload() function. The callParams encode the refund address as the msg.sender, which in this case would be the DecentBridgeAdapter contract if the call is made from the bridgeAndExecute() function in the UTB contract.

```
File: DecentEthRouter.sol
        ICommonOFT.LzCallParams memory callParams = ICommonOFT.LzCallParams({
            refundAddress: payable(msg.sender),
            zroPaymentAddress: address(0x0),
            adapterParams: adapterParams
        });
```

2. Due to this, if any excess gas is provided to the sendAndCall() operation, it will be refunded to the adapter instead of the user. This excess gas amount would then be permanently locked inside the DecentBridgeAdapter contract.

```
File: DecentEthRouter.sol
        dcntEth.sendAndCall{value: gasValue}(
            address(this), // from address that has dcntEth (so DecentRouter)
            _dstChainId,
            destinationBridge, // toAddress
            _amount, // amount
            payload, //payload (will have recipients address)
            _dstGasForCall, // dstGasForCall
            callParams // refundAddress, zroPaymentAddress, adapterParams
        );
```

Tools Used

Manual Review

Recommended Mitigation Steps

Use refund address as the user who initiated the bridgeAndExecute() transaction.