Zellic

Ulysses Protocol

Smart Contract Security Assessment

May 23, 2023

Prepared for:

Maia DAO

Prepared by:

Ulrich Myhre and Katerina Belotskaia

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Maia DAO from March 27th to April 14th, 2023. During this engagement, Zellic reviewed Ulysses Protocol's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is it possible to obtain and exploit reentrancy in the ports or in a bridge agent?
- Can the bridge agent contract be DDOSed through anycalls?
- Are there exploitable issues related to overflow in the bandwidth increase or rounding issues in the balance calculations?
- Is cross-chain gas management working correctly?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside of the assessment scope

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, a lack of internal documentation prevented us from verifying some of the business logic. The large codebase, combined with the amount of findings and broken functionality, left less time for writing a full threat model of every external function. Instead, the time was prioritized on analyzing the code and studying the various findings.

1.3 Results

During our assessment on the scoped Ulysses Protocol contracts, we discovered 18 findings. Five critical issues were found. Of the other findings, five were of high impact, three were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Maia DAO's benefit in the Discussion section (4) at the end of the document.

Summary

Overall, our assessment of the code is that it is currently incomplete and requires improvement in the following areas before deployment:

1. test code coverage
2. function-level documentation
3. robustness of the protocol.

Test code coverage is insufficient, and there is a need to add unit tests and integration tests for various user interactions. Several functions in the codebase are currently not functioning correctly and appear to be untested. For example, the `sweep` function in RootBridgeAgent is not retrieving any fees, while the `anyExecuteSettlement` function in CoreBrachRouter has two branches that share the same constant. These issues indicate a lack of testing and validation, which can potentially lead to critical bugs and security vulnerabilities.

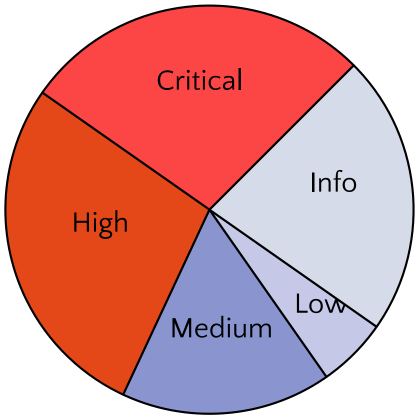
While the documentation provides a helpful overview of the contract ecosystem, we found that the Technical Reference section could be improved. The section currently states that it is under construction and provides no information about the smart contracts. This could be frustrating for users trying to understand the technical details and the roles and responsibilities of various actors in the system. We recommend that the project team prioritize the completion of the Technical Reference section and ensure that it provides clear and accurate information about the smart contracts. Additionally, we found that some of the function-level documentation was copied from unrelated sources without updating the text, which can be confusing for users. We recommend reviewing and updating all function-level documentation to ensure clarity and accuracy.

We also propose enhancing the robustness of the protocol by developing a thorough test suite. As discussed in 4.2, there are further suggestions related to test suite improvement that we encourage the project team to consider.

To ensure the protocol is as secure and reliable as possible, we strongly suggest conducting a comprehensive reaudit upon completion of development and before deployment of the protocol's contracts. A reaudit is valuable in identifying any potential issues or vulnerabilities and allowing for any necessary changes or fixes to be made before deployment.

Breakdown of Finding Impacts

Impact Level	Count
Critical	5
High	5
Medium	3
Low	1
Informational	4



2 Introduction

2.1 About Ulysses Protocol

Ulysses Protocol is a decentralized and permissionless community-owned omnichain liquidity protocol designed to promote capital efficiency in the face of an evermore fragmented liquidity landscape in DeFi. It enables liquidity providers to deploy their assets from one chain and earn revenue from activity in an array of chains all while mitigating the negative effects of current market solutions. In addition, it offers a way for DeFi protocols to lower their operational and managing costs by incentivizing liquidity for a single unified liquidity token instead of managing incentives for pools scattered across different AMMs and chains.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Ulysses Protocol Contracts

Repository https://github.com/Maia-DAO/maia-ecosystem-monorepo

Version maia-ecosystem-monorepo: `322a25688c9ce2ce3c30e7d794c475e32447 1a2f`

Programs
- ERC4626
- ERC4626DepositOnly
- ERC4626MultiToken
- UlyssesERC4626
- IERC4626
- IERC4626DepositOnly
- IERC4626MultiToken

- IUlyssesERC4626
- UlyssesPool
- UlyssesRouter
- UlyssesToken
- UlyssesFactory
- IUlyssesFactory
- IUlyssesPool
- IUlyssesRouter
- IUlyssesToken
- ArbitrumBranchBridgeAgent
- ArbitrumBranchPort
- ArbitrumCoreBranchRouter
- BaseBranchRouter
- BasePortGauge
- BranchBridgeAgent
- BranchPort
- CoreBranchRouter
- CoreRootRouter
- MulticallRootRouter
- RootBridgeAgent
- RootPort
- VirtualAccount
- ArbitrumBranchBridgeAgentFactory
- BranchBridgeAgentFactory
- ERC20hTokenBranchFactory
- ERC20hTokenRootFactory
- RootBridgeAgentFactory

- BytesLib
- IAnycallConfig
- IAnycallExecutor
- IAnycallProxy
- IApp
- IArbBranchPort
- IBranchBridgeAgent
- IBranchBridgeAgentFactory
- IBranchPort
- IBranchRouter
- ICoreBranchRouter
- ICoreBridgeAgent
- ICoreRootBridgeAgent
- IERC20hTokenBranch
- IERC20hTokenBranchFactory
- IERC20hTokenRoot
- IERC20hTokenRootFactory
- IERC721Permit
- IFeePool
- IMulticall2
- INonfungiblePositionManager
- IPeripheryImmutableState
- IPeripheryPayments
- IPoolInitializer
- IPortStrategy
- IRootBridgeAgent
- IRootBridgeAgentFactory

- IRootPort
- IRootRouter
- ISwapRouter
- IUniswapV3SwapCallback
- IVirtualAccount
- IWETH9
- Path
- PoolAddress
- AnycallFlags
- CoreRouterLib
- Multicall2
- ERC20hTokenBranch
- ERC20hTokenRoot

Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of five person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Ulrich Myhre, Engineer
unblvr@zellic.io

Katerina Belotskaia, Engineer
kate@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

March 27, 2023	Kick-off call
March 27, 2023	Start of primary review period
April 14, 2023	End of primary review period
April 27, 2023	Draft report delivered
May 23, 2023	Final report delivered

3 Detailed Findings

3.1 Lack of the RootBridgeAgent contract verification

- **Target**: ArbitrumBranchBridgeAgentFactory, ArbitrumBranchBridgeAgent
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The function `createBridgeAgent` from ArbitrumBranchBridgeAgentFactory allows any caller to create a new ArbitrumBranchBridgeAgent contract with controlled `_newBranchRouterAddress` and `_rootBridgeAgentAddress`.

```
function createBridgeAgent(
        address _newBranchRouterAddress,
        address _rootBridgeAgentAddress
    ) external override returns (address newBridgeAgent) {
        newBridgeAgent = address(
            new ArbitrumBranchBridgeAgent(
                wrappedNativeToken,
                rootChainId,
                _rootBridgeAgentAddress,
                localAnyCallAddress,
                localAnyCallExecutorAddress,
                _newBranchRouterAddress,
                localPortAddress
            )
        );

        IPort(localPortAddress).addBridgeAgent(newBridgeAgent);
    }
```

After that, the `addBridgeAgent` function of the localPortAddress smart contract is called, passing in the newly created contract's address `newBridgeAgent` as a parameter. After adding the address, the BridgeAgent contract will have access to functions with the `requiresBridgeAgent` modifier to call: `depositToPort`, `withdrawFromPort`, `withdraw`, `bridgeIn`, `bridgeInMultiple`, `bridgeOut`, and `bridgeOutMultiple`.

For the `depositToPort` and `withdrawFromPort` functions, it is not possible to manipulate parameters:

```
function withdrawFromPort(address localAddress, uint256 amount)
    external payable lock {
        IArbPort(localPortAddress).withdrawFromPort(msg.sender,
    msg.sender, localAddress, amount);
    }
```

But, for example, for the `bridgeIn` and `withdraw` functions, all parameters are controlled by the `rootBridgeAgentAddress`, which is allowed to call `anyExecute` (as you can see in the provided code below):

```
contract BranchBridgeAgent is IBranchBridgeAgent {
     ...
    function _requiresExecutor() internal view override {
        if (msg.sender ≠ rootBridgeAgentAddress)
    revert AnycallUnauthorizedCaller();
    }

    function anyExecute(bytes calldata data)
        external
        virtual
        requiresExecutor
        returns (bool success, bytes memory result)
    {
        //[ ... Execute Calldata ... ]
}
```

The `_withdraw` function allows to transfer `_underlyingAddress` tokens from `localPortAddress` to `_recipient`. All parameters are controlled by the caller, and therefore the `rootBridgeAgentAddress` has full control.

```
contract ArbitrumBranchPort is BranchPort, IArbBranchPort {
     ...
    function withdraw(address _recipient, address _underlyingAddress,
    uint256 _amount)
        external
        override(IBranchPort, BranchPort)
        requiresBridgeAgent
    {
        _withdraw(_recipient, _underlyingAddress, _amount);
    }
    function _withdraw(address _recipient, address _underlyingAddress,
    uint256 _amount) internal override(BranchPort) {
        _underlyingAddress.safeTransfer(_recipient, _amount);
    }
    ...
}
```

The `bridgeIn` function allows to mint an arbitrary amount of `_hToken` tokens for the `_recipient`.

```
contract ArbitrumBranchPort is BranchPort, IArbBranchPort {
    ...
    function bridgeIn(address _recipient, address _localAddress,
    uint256 _amount)
        external
        override(IBranchPort, BranchPort)
        requiresBridgeAgent
    {
        IRootPort(rootPortAddress).mintToLocalBranch(_recipient,
    _localAddress, _amount);
    }
    ...
}
    contract RootPort is Ownable, IRootPort {
    ...
    function mintToLocalBranch(address _recipient, address _hToken,
    uint256 _amount) external requiresLocalBranchPort {
        mint(_recipient, _hToken, _amount, localChainId);
    }
    function mint(address _to, address _hToken, uint256 _amount,
    uint24 _fromChain) internal {
        ERC20hTokenRoot(_hToken).mint(_to, _amount, _fromChain);
    }
    ...
}
```

Impact

Since the attacker has full control over the rootBridgeAgentAddress contract, they have no restrictions on calling the `anyExecute` function and therefore have full control over the parameters and can withdraw all funds from the `localPortAddress` contract as well as mint any number of tokens through the RootPort contract.

Recommendations

It is important to add additional validation to ensure the `_rootBridgeAgentAddress` contract address is legitimate and has been created securely through the RootBridgeAgentFactory contract.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit 92ef9cce.

In the remediation phase, Maia DAO made extensive changes to address the finding, resulting in a substantial amount of code modifications. The diff between the audited code and the commit containing the fixes was significant, encompassing over 3000 lines of code. While we focused our review on the specific fix, it is important to note that the extensive changes made it challenging to thoroughly audit all aspects of the code affected by the fix. Although we believe the fix addresses the initial finding, there is a possibility that new bugs or vulnerabilities may have been introduced due to the expansive nature of the changes.

To ensure the overall security and stability of the system, we recommend a comprehensive reaudit of the codebase to thoroughly assess the changes made and identify any potential issues that may have arisen.

3.2 Missing access control for Multicall

- **Target**: MulticallRootRouter
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The function `anyExecuteSignedDepositMultiple` lacks the `requiresAgent` modifier, which makes it callable by anyone.

```
function anyExecuteSignedDepositMultiple(
    bytes1 funcId,
    bytes memory rlpEncodedData,
    DepositMultipleParams calldata,
    address userAccount,
    uint24 fromChainId
) external payable returns (bool success, bytes memory result)
{ ... }
```

Depending on the `funcId`, multiple things can happen, but `IVirtualAccount(userAccount).call(calls)` is always called on the `userAccount` input.

If this is an actual VirtualAccount implementation, the `call()` function will revert, since it is protected by a `requiresApprovedCaller` modifier, and this approval is toggled on and off by `RootBridgeAgent.sol` calling `IPort(localPortAddress).toggleVirtualAccountApproved( ... )` before and after the call to `anyExecuteSignedDepositMultiple`. An attacker can pick their own contract that pretends to be a VirtualAccount and make calls to, for example, `call( ... )` or `withdrawERC20( ... )` successful.

This in itself is not helpful for an attacker, but for `funcId` `0x02` and `0x03`, there are calls to the internal functions `_approveAndCallOut( ... )` and `_approveMultipleAndCallOut( ... )`. The attacker controls all parameters going into these functions. This ends up transferring tokens from Root to Branch, then sending or minting money to the receiver.

Impact

Using a fake VirtualAccount contract, a user can steal tokens from the Root by directly calling `anyExecuteSignedDepositMultiple( ... )`. Note that this full chain is hard to verify because it relies on several encoded structures and dependencies, so there is no proof of concept for this attack.

Recommendations

Add a `requiresAgent` modifier to the `anyExecuteSignedDepositMultiple` function.

Remediation

This issue has been acknowledged by Maia DAO, and fixes were implemented in the following commits:

- ca057685
- 42c35522

3.3 Multitoken lacks validations

- **Target**: ERC4626MultiToken
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

Several functions within ERC4626MultiToken.sol lack necessary validations, which can result in the loss of funds or broken contracts. This contract is similar to the Solmate implementation of ERC4626, but it differs in that it allows for the trading of multiple (weighted) assets instead of just one. This is implemented by replacing the `uint256 assets` argument with a `uint256[] memory assetsAmounts` array.

The `deposit` function calls multiple functions, each of which iterates based on the length of the `assetsAmounts` input array. However, **this array is never checked to ensure that it is equal to `assets.length`**. The function then calculates the amount of shares by calling `previewDeposit(assetsAmounts)`, which is a wrapper for `convertToShares`.

```
function convertToShares(uint256[] memory assetsAmounts)
    public view virtual returns (uint256 shares) {
        uint256 _totalWeights = totalWeights;
        uint256 length = assetsAmounts.length;

        shares = type(uint256).max;
        for (uint256 i = 0; i < length;) {
            uint256 share = assetsAmounts[i].mulDiv(_totalWeights,
    weights[i]);
            if (share < shares) shares = share;
            unchecked {
                i++; // @audit ++i
            }
        }
    }
```

Here, the `shares` variable is calculated based on the smallest possible `share = assetsAmounts[i] * _totalWeights / weights[i]`. After this, the `receiveAssets(assetsAmounts)` function is called to actually transfer the assets to the contract. **However, if the `assetsAmounts.length == 0`, `shares` will be `type(uint256).max`**. Lastly, it mints the amount of shares and awards this to the receiver.

Upon calling the `redeem` function, a user can present their shares and get back a mix of assets based on the weights, despite only depositing a subset of the assets (or none at all).

Additionally, the `constructor` does not verify that weights are nonzero nor that the length of `assets` and `weights` are equal.

A test case that proves this behavior was implemented inside `UlyssesTokenHandler.t.sol`,

```
function test_poc_deposit() public virtual {
    address addr = 0xaAaAaAaaAaAaAaaAaAAAAAAAAaaaAaAaAaaAaaAa;
    uint[] memory assets;
    console.log(UlyssesToken(_vault_).deposit(assets, addr));
}
```

where the output is

```
Running 1 test for test/2-audit/ulysses-
    amm/UlyssesTokenTest.t.sol:InvariantUlyssesToken
[PASS] test_poc_deposit() (gas: 62631)
Logs:
11579208923731619542357098500868790785326998466564056403945758400791312963
39935

Test result: ok. 1 passed; 0 failed; finished in 1.19ms
```

This shows that one can mint infinite shares without depositing any assets.

Impact

- If a zero-length `assetsAmounts` is allowed, users can obtain infinite shares for free. This lets them drain the contract.
- Since the lengths of `assetsAmounts` and `assets` are not synchronized, it is possible to add a single asset, get shares, then redeem multiple assets after.
- Since the lowest amount of shares is picked, a user that sends an uneven amount of tokens could get less shares than expected. This can be exacerbated by the weights changing before the transaction is included in the block, where there is no slippage protection parameter.

For the constructor validation issues, the contract will break if one of the weights are zero, as it divides by the weight when withdrawing.

Recommendations

Check that `assetsAmounts` and `assets` have the same length in every location and disallow empty arrays where it can skip important loops. Add validation in the `constructor` to ensure that the contract cannot be initialized in a way that breaks functionality in the future.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit df6b941b.

3.4 Multiple redeems of the same deposit are possible

- **Target**: BranchBridgeAgent
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `redeemDeposit` function currently allows any caller to initiate the withdrawal of a deposit that is in a `Failed` status. The caller has control over the associated `_depositNonce` value, which includes information such as the `hToken` addresses and amounts, the `underlying` addresses and amounts of deposited tokens, and the owner of the deposit who will receive the funds.

Once the funds have been successfully withdrawn, the deposit data is not reset or modified in any way, which means that it is possible to call the function again with the same identifier.

```
function redeemDeposit(uint32 _depositNonce) external lock {
        //Update Deposit
        if (getDeposit[_depositNonce].status != DepositStatus.Failed) {
            revert DepositRedeemUnavailable();
        }
        _redeemDeposit(_depositNonce);
    }

    function _redeemDeposit(uint32 _depositNonce) internal {
        //Get Deposit
        Deposit storage deposit = _getDepositEntry(_depositNonce);

        //Transfer token to depositor / user
        for (uint256 i = 0; i < deposit.hTokens.length;) {
            if (deposit.amounts[i] - deposit.deposits[i] > 0) {
                IPort(localPortAddress).bridgeIn(
                    deposit.owner, deposit.hTokens[i], deposit.amounts[i]
    - deposit.deposits[i]
                );
            }
            IPort(localPortAddress).withdraw(deposit.owner,
    deposit.tokens[i], deposit.deposits[i]);

            unchecked {
                ++i;
            }
        }
        IPort(localPortAddress).withdraw(deposit.owner,
    address(wrappedNativeToken), deposit.depositedGas);
    }
```

Impact

If a cross-chain call fails, and the status of `deposit` will be set to `Failed`, the depositor will be able to withdraw all the `_underlyingAddress` tokens deposited to the localPortAddress contract.

Recommendations

After a successful withdrawal, update the status of the deposit and reset the amounts of deposited funds or delete the deposit information from storage altogether. This will help prevent any repeated withdrawal of funds and ensure that the contract state accurately reflects the state of the deposits.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit a0dd0311.

3.5 Broken fee sweep

- **Target**: RootBridgeAgent
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: Critical

Description

Whenever execution gas is paid in `_payExecutionGas( ... )`, a fee is taken and stored in the global `accumulatedFees`. To withdraw these fees, the function `sweep()` is called by the designated `daoAddress`, and the fees are then supposed to be reset afterwards.

```
function sweep() external {
        if (msg.sender ≠ daoAddress) revert UnauthorizedCaller();
        accumulatedFees = 0;
        SafeTransferLib.safeTransferETH(daoAddress, accumulatedFees);
    }
```

However, `accumulatedFees` is reset *before* the token transfer.

Impact

Fees are stuck in the RootBridgeAgent. It is impossible to extract these from the contract.

Recommendations

Move the `accumulatedFees` reset below the transfer call, but also consider the need for reentrancy guards.

Remediation

A temporary variable was introduced to mirror the accumulated fees, and the global is still reset before the transfer to avoid reentrancy guards. This issue has been acknowledged by Maia DAO, and a fix was implemented in commit 23c47122.

3.6 Asset removal is broken

- **Target**: UlyssesToken.sol
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The function `removeAsset(address asset)` removes an asset from the global `assets[]` and `weights[]` arrays and updates some globals.

```
function removeAsset(address asset) external nonReentrant onlyOwner {
        // No need to check if index is 0, it will underflow and revert if
    it is 0
        uint256 assetIndex = assetId[asset] - 1;

        if (assets.length == 1) revert CannotRemoveLastAsset();

        // Remove asset from array
        for (uint256 i = assetIndex; i < assets.length; i++) {
            assets[i] = assets[i + 1];
            weights[i] = weights[i + 1];
        }

        totalWeights -= weights[assetIndex];

        assets.pop();
        weights.pop();
        assetId[asset] = 0;
     ...
    }
```

This is done by looking up the index of the asset in `assetId`, then moving all assets and weights down by one index. Finally, `totalweights` is supposed to be reduced by the weight of the removed asset, and the duplicated value at the end is popped off. However, there are multiple issues with this implementation.

- The loop increments `i` to `assets.length` but indexes into `i+1`, which will go beyond the length of the array and revert.
- Global `totalWeights` is reduced *after* the target asset and weight has been overwritten, reducing the `totalWeights` by a different weight than intended.

- The `assetId` mapping is supposed to point to the index of a given asset, but these indices ares not updated when all the positions shift around.
- While not unsolveable, adding too many assets can make it impossible to remove one of the lower-index assets due to gas cost. Removing higher-index assets would be possible still, and multiple such operations could reduce the gas cost for a lower index too.

Impact

It is impossible to remove assets. Even if it worked, the weights would be wrong after removing an asset. Due to `assetId` not updating, removing an asset in the future will remove the wrong asset — or cause the transaction to revert.

Recommendations

- Loop to `assets.length-1`.
- Update `totalWeights` before removing the weights.
- Update the `assetId` mapping.
- Create test cases for the function.

Remediation

This issue has been acknowledged by Maia DAO, and fixes were implemented in the following commits:

- 3d317ac6
- f116e00e

3.7 Unsupported function codes

- **Target**: CoreBranchRouter
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

Several functions from CoreBranchRouter perform an external call to `IBridgeAgent(localBridgeAgentAddress).performSystemCallOut`; below is an example of this kind of call. The `data` generated from user input is encoded with a byte responsible for the type of function to be executed as a result of cross-chain communication. In this case, the byte `0x01` is responsible for adding a new global token.

```
contract CoreBranchRouter is BaseBranchRouter {
     ...
    function addGlobalToken(
        address _globalAddress,
        uint256 _toChain,
        uint128 _remoteExecutionGas,
        uint128 _rootExecutionGas
    ) external payable {
        bytes memory data = abi.encode(address(this), _globalAddress,
    _toChain, _rootExecutionGas);
        bytes memory packedData = abi.encodePacked(bytes1(0x01), data);

    IBridgeAgent(localBridgeAgentAddress).performSystemCallOut{value:
    msg.value}(
            msg.sender, packedData, _remoteExecutionGas
        );
    }
     ...
}
```

The `performSystemCallOut` function encodes the user's data using a byte `0x00` that determines the type of function to be called within the RootBridgeAgent contract. The resulting encoded `data` is then passed for execution.

```
function performSystemCallOut(address depositor, bytes calldata params,
    uint128 rootExecutionGas)
```

```
        external
        payable
        lock
        requiresRouter
        requiresFallbackGas
    {
        bytes memory data =
            abi.encodePacked(bytes1(0x00), depositNonce, params,
    msg.value.toUint128(), rootExecutionGas);
        _depositAndCall(depositor, data, address(0), address(0), 0, 0); //
    → IRootBridgeAgent(rootBridgeAgentAddress).anyExecute(_callData);
    }
```

During execution, the `data` will be decoded in the `anyExecute` function. The `0x00` byte corresponds to the execution of the `IRouter(localRouterAddress).anyExecuteResponse` function, which will be called with the decoded `data`.

```
contract RootBridgeAgent is IRootBridgeAgent {
    function anyExecute(bytes calldata data)
        external
        virtual
        requiresExecutor
        returns (bool success, bytes memory result)
    {
        ...
        bytes1 flag = data[0];
        if (flag == 0x00) {

    IRouter(localRouterAddress).anyExecuteResponse(bytes1(data[5]),
    data[6:data.length - PARAMS_GAS_IN], fromChainId);
        } else if (flag == 0x01) {
            IRouter(localRouterAddress).anyExecute(bytes1(data[5]),
    data[6:data.length - PARAMS_GAS_IN], fromChainId);
            emit LogCallin(flag, data, fromChainId);
        }
        ...
    }
    ...
}
```

But the current implementation of `localRouterAddress.anyExecuteResponse` supports only the `0x02` and `0x03` function IDs. Therefore, for the above example with `addGlobalToken` (`0x01` funcId), the `anyExecuteResponse` will return a `false` status with an `unknown selector` message.

```
contract CoreRootRouter is IRootRouter, Ownable {
     ...
    function anyExecuteResponse(bytes1 funcId,
    bytes calldata encodedData, uint24 fromChainId)
        external
        payable
        override
        requiresAgent
        returns (bool, bytes memory)
    {
        /// FUNC ID: 2 (_addLocalToken)
        if (funcId == 0x02) {
            (address underlyingAddress, address localAddress,
    string memory name, string memory symbol) =
                abi.decode(encodedData, (address, address, string,
    string));

            _addLocalToken(underlyingAddress, localAddress, name, symbol,
    fromChainId);

            emit LogCallin(funcId, encodedData, fromChainId);
            /// FUNC ID: 3 (_setLocalToken)
        } else if (funcId == 0x03) {
            (address globalAddress, address localAddress)
    = abi.decode(encodedData, (address, address));

            _setLocalToken(globalAddress, localAddress, fromChainId);

            emit LogCallin(funcId, encodedData, fromChainId);

            /// Unrecognized Function Selector
        } else {
            return (false, "unknown selector");
        }
        return (true, "");
    }
     ...
}
```

Another example of a function from `CoreBranchRouter` that cannot be executed is `syncBridgeAgent`. This function corresponds to an identifier `0x04` that is also not supported by `anyExecuteResponse`.

```
contract CoreBranchRouter is BaseBranchRouter {
     ...
    function syncBridgeAgent(address _newBridgeAgentAddress,
    address _rootBridgeAgentAddress) external payable {
        if
    (!IPort(localPortAddress).isBridgeAgent(_newBridgeAgentAddress))
    {
            revert UnrecognizedBridgeAgent();
        }
        bytes memory data = abi.encode(_newBridgeAgentAddress,
    _rootBridgeAgentAddress);
        bytes memory packedData = abi.encodePacked(bytes1(0x04), data);

    IBridgeAgent(localBridgeAgentAddress).performSystemCallOut{value:
    msg.value}(msg.sender, packedData,
    0);
    }
     ...
}
```

Impact

Calling functions that are not supported by the final executor can result in the loss of funds paid for gas and can disrupt the operation of user applications waiting for successful cross-chain function execution.

Recommendations

The `anyExecute` function supports function IDs `0x04` and `0x01`. In order to call this function as a result of cross-chain communication, the `addGlobalToken` and `syncBridgeAgent` functions should call `IBridgeAgent(localBridgeAgentAddress).performCallOut` instead of `IBridgeAgent(localBridgeAgentAddress).performSystemCallOut`.

This ensures that the `0x01` flag is encapsulated in the `data`, allowing the `IRouter(localRouterAddress).anyExecute` function to be called upon decoding.

```
function anyExecute(bytes1 funcId, bytes calldata encodedData,
    uint24 fromChainId)
        external
        payable
        override
        requiresAgent
        returns (bool, bytes memory)
    {
        /// FUNC ID: 1 (_addGlobalToken)
        if (funcId == 0x01) {
            (address branchRouter, address globalAddress, uint24 toChain,
    uint128 remoteExecutionGas) =
                abi.decode(encodedData, (address, address, uint24,
    uint128));

            _addGlobalToken(remoteExecutionGas, globalAddress,
    branchRouter, toChain);

            emit LogCallin(funcId, encodedData, fromChainId);

            /// FUNC ID: 4 (_syncBranchBridgeAgent)
        } else if (funcId == 0x04) {
            (address newBranchBridgeAgent, address rootBridgeAgent)
    = abi.decode(encodedData, (address, address));

            _syncBranchBridgeAgent(newBranchBridgeAgent, rootBridgeAgent,
    fromChainId);

            emit LogCallin(funcId, encodedData, fromChainId);

            /// Unrecognized Function Selector
        } else {
            return (false, "unknown selector");
        }
        return (true, "");
    }
```

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit 92ef9cce.

3.8 Missing access control on `anyExecuteNoSettlement`

- **Target**: CoreBranchRouter
- **Category**: Coding Mistakes
- **Severity**: High
- **Likelihood**: High
- **Impact**: High

Description

The `anyExecuteNoSettlement` function is responsible for executing a cross-chain request. It is supposed to be called from the `anyExecute` function in the BranchBridgeAgent contract. But since the function has no caller verification, anyone can execute it.

```
function anyExecuteNoSettlement(bytes memory _data)
        external
        virtual
        override
        returns (bool success, bytes memory result)
    {
        if (_data[0] == 0x01) {
            (, address globalAddress, string memory name,
    string memory symbol, uint128 gasToBridgeOut) =
                abi.decode(_data, (bytes1, address, string, string,
    uint128));

            _receiveAddGlobalToken(globalAddress, name, symbol,
    gasToBridgeOut);

            /// Unrecognized Function Selector
        } else if (_data[0] == 0x01) { //@audit unreachable branch
            (, address newBridgeAgentFactoryAddress) = abi.decode(_data,
    (bytes1, address));

            _receiveAddBridgeAgentFactory(newBridgeAgentFactoryAddress);

            /// Unrecognized Function Selector
        } else {
            return (false, "unknown selector");
        }
        return (true, "");
    }
```

Note that there is an error in the current implementation of the `anyExecuteNoSettlement` function that does not allow calling the function `_receiveAddBridgeAgentFactory`, since the else if branch cannot be executed. This is because both branches check the equality of `_data[0]` to `0x01`, and the first if will be executed in priority.

Impact

Since there is no caller verification on the `anyExecuteNoSettlement` function, any caller is able to execute the `_receiveAddGlobalToken` function, manipulate the input parameters `globalAddress, name, symbol, gasToBridgeOut` and pass them to the `performSystemCallOut` function, which performs a call to the AnycallProxy contract for cross-chain messaging.

```
function _receiveAddGlobalToken(
        address _globalAddress,
        string memory _name,
        string memory _symbol,
        uint128 _rootExecutionGas
    ) internal {
        //Create Token
        ERC20hToken newToken
    = IFactory(hTokenFactoryAddress).createToken(_name, _symbol);

        //Encode Data
        bytes memory data = abi.encode(_globalAddress, newToken);

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

        //Send Cross-Chain request

    IBridgeAgent(localBridgeAgentAddress).performSystemCallOut{value:
    _rootExecutionGas}(
            address(this), packedData, 0
        );
    }
```

Next, as a result of cross-chain communication, the `anyExecuteResponse` function will be executed and the `globalAddress`, controlled by the `anyExecuteNoSettlement` caller, will be passed to the `IPort(rootPortAddress).setLocalAddress` function.

```
contract CoreRootRouter is IRootRouter, Ownable {
     ...
    function anyExecuteResponse(bytes1 funcId,
    bytes calldata encodedData, uint24 fromChainId)
        external
        payable
        override
        requiresAgent
        returns (bool, bytes memory)
    {
      ...
      } else if (funcId == 0x03) {
            (address globalAddress, address localAddress)
    = abi.decode(encodedData, (address, address));

            _setLocalToken(globalAddress, localAddress, fromChainId);
        ...
    }

    function _setLocalToken(address _globalAddress,
    address _localAddress, uint24 _toChain) internal {
        IPort(rootPortAddress).setLocalAddress(_globalAddress,
    _localAddress, _toChain);
    }
}
```

The setLocalAddress function currently allows modifications to the getGlobalAddressFromLocal and getLocalAddressFromGlobal mappings without any checks. This means that the existing getLocalAddressFromGlobal[_fromChain][_globalAddress] value can be overwritten.

```
function setLocalAddress(address _globalAddress, address _localAddress,
    uint24 _fromChain)
        external
        requiresCoreBridgeAgent
    {
        getGlobalAddressFromLocal[_fromChain][_localAddress]
    = _globalAddress;
        getLocalAddressFromGlobal[_fromChain][_globalAddress]
    = _localAddress;
    }
```

Recommendations

The `requiresBridgeAgent` modifier should be used to prevent anyone from calling the `anyExecuteNoSettlement` function.

Remediation

The `requiresBridgeAgent` modifier was added in commit c73b4c5d.

The implementation of the `anyExecuteNoSettlement` function was fixed in commits d588989e and 92ef9cce.

3.9 The protocol fee from pools will be claimed to zero address

- **Target**: UlyssesFactory
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The UlyssesFactory contract enables the creation of new pools contracts and sets its own address as the factory address. Protocol fees collected from the pool contracts are transferred to the owner of the factory. However, the `_initializeOwner` function is not called in the factory contract, resulting in the `factory.owner()` returning `address(0)`.

```
function claimProtocolFees()
    external nonReentrant onlyOwner returns (uint256 claimed) {
        claimed = getProtocolFees();

        if (claimed > 0) {
            asset.safeTransfer(factory.owner(), claimed);
        }
    }
```

Impact

Since the owner of the UlyssesFactory contract is not set during its creation, it will not be possible to change the owner at a later time. This means that it will also not be possible to withdraw the protocol fee from pool contracts, as the `factory.owner()` function will return the zero address `address(0)`.

In addition, the fee amount cannot be changed, because the `setProtocolFee` is allowed to be called only by `factory.owner()`.

```
function setProtocolFee(uint256 _protocolFee) external nonReentrant {
        if (msg.sender ≠ factory.owner()) revert Unauthorized();

        // Revert if the protocol fee is larger than 1%
        if (_protocolFee > MAX_PROTOCOL_FEE) revert InvalidFee();

        protocolFee = _protocolFee;
    }
```

Recommendations

Pass the owner's address to the `UlyssesFactory` constructor and set the owner using the `_initializeOwner` function.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit bd2054cb.

3.10 Unlimited cross-chain asset transfer without deposit requirement

- **Target**: RootBridgeAgent
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The `callOutAndBridge` function facilitates the transfer of assets from the root chain to the branch omnichain environment and creates a `Settlement` object that contains information about the amount of tokens and the addresses involved in the transfer.

In the event that the cross-chain call fails or reverts, the `anyFallback` function is called to reopen the `Settlement` object and set its status to `Pending`. This enables a retry through the `clearSettlement` function.

```
function clearSettlement(uint32 _settlementNonce,
    uint128 _remoteExecutionGas) external payable {
        //Update User Gas available.
        if (initialGas == 0) {
            userFeeInfo.depositedGas = uint128(msg.value);
            userFeeInfo.gasToBridgeOut = _remoteExecutionGas;
        }
        //Clear Settlement with updated gas.
        _clearSettlement(_settlementNonce);
    }
```

The `clearSettlement` function initiates resending of a failed cross-chain transfer by calling the internal `_clearSettlement` function. The status of the settlement object is checked to ensure that it is currently in a `Pending` state before attempting to resend. If the resend is successful, the status of the temporary `settlement` variable is set to `Success`. However, it is important to note that the status of the actual `Settlement` object is not changed during the execution of this function and will remain as `Pending`.

```
function _clearSettlement(uint32 _settlementNonce)
    internal requiresFallbackGas {
        //Get settlement
        Settlement memory settlement
    = _getSettlementEntry(_settlementNonce);
```

```
        //Require Status to be Pending
        require(settlement.status == SettlementStatus.Pending);

        //Update Settlement
        settlement.status = SettlementStatus.Success;

        //Slice last 4 bytes calldata
        uint128 prevGasToBridgeOut =
            uint128(bytes16(BytesLib.slice(settlement.callData,
    settlement.callData.length - 16, 16)));

        //abi encodePacked
        bytes memory newGas = abi.encodePacked(prevGasToBridgeOut
    + _manageGasOut(settlement.toChain));

        //overwrite last 16bytes of callData
        for (uint256 i = 0; i < newGas.length;) {
            settlement.callData[settlement.callData.length - 16 + i]
    = newGas[i];
            unchecked {
                ++i;
            }
        }
        //Set Settlement
        getSettlement[_settlementNonce].callData = settlement.callData;
        //Retry call with additional gas
        _performCall(settlement.callData, settlement.toChain);
    }
```

Impact

Users will be able to move assets from the root chain to the branch omnichain environment without depositing additional funds.

Recommendations

We recommend implementing the status change as shown below:

```
function _clearSettlement(uint32 _settlementNonce)
    internal requiresFallbackGas {
        //Get settlement
        Settlement memory settlement
    = _getSettlementEntry(_settlementNonce);

        //Require Status to be Pending
        require(settlement.status == SettlementStatus.Pending);

        //Update Settlement
        settlement.status = SettlementStatus.Success;

    getSettlement[_settlementNonce].status = SettlementStatus.Success;
         ...
```

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit 073012d1.

3.11 ChainId type confusion

- **Target**: CoreBranchRouter, RootBridgeAgent, RootPort
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

To distinguish between the various chains being used, each chain is assigned a unique identifier that determines where cross-chain calls will be executed. Throughout the code, the chain identifier is represented as both uint256 and uint24, which can create issues when data is packed and unpacked without an explicit cast. This occurs in several locations, as illustrated by the following examples:

CoreBranchRouter

`CoreBranchRouter.sol` → `addGlobalToken( ... )` has `_toChain` represented as uint256. These parameters are then encoded with `abi.encode` and passed down to be called cross chain.

```
function addGlobalToken(
    address _globalAddress,
    uint256 _toChain,
    uint128 _remoteExecutionGas,
    uint128 _rootExecutionGas
) external payable {
    //Encode Call Data
    bytes memory data = abi.encode(address(this), _globalAddress,
    _toChain, _rootExecutionGas);
    ...
}
```

It ends up here, in `CoreRootRouter→anyExecute`,

```
function anyExecute(bytes1 funcId, bytes calldata encodedData,
    uint24 fromChainId)
    external
    payable
    override
    requiresAgent
    returns (bool, bytes memory)
{
    /// FUNC ID: 1 (_addGlobalToken)
    if (funcId == 0x01) {
        (address branchRouter, address globalAddress, uint24 toChain,
    uint128 remoteExecutionGas) =
            abi.decode(encodedData, (address, address, uint24, uint128));
    ... }
 ...
}
```

where it is decoded with `abi.decode` as a uint24. Since the non-packed version of `abi.encode` is used, this will work until `_toChain` some day is picked to be too large to represented as uint24, and then the `anyExecute` call will start to revert.

RootBridgeAgent

The function `_payExecutionGas( ... , uint24 _fromChain, uint256 _toChain)` uses both uint24 and uint256 to represent chains in the same function signature. However, other functions tend to use uint24 to represent it, including functions that do slicing of the input parameters by directly accessing the bytes.

RootPort

The internal function `_getLocalToken` looks up a local token address on a different chain,

```
function _getLocalToken(address _localAddress, uint256 _fromChain,
    uint24 _toChain)
    internal
    view
    returns (address)
{
    address globalAddress
    = getGlobalAddressFromLocal[_fromChain][_localAddress];
    return getLocalAddressFromGlobal[_toChain][globalAddress];
}
```

where `_toChain` is a uint24, but the mapping `getLocalAddressFromGlobal` is defined as `mapping(uint256 => mapping(address => address)) public getLocalAddressFromGlobal`.

Impact

Compilers are not able to effectively reason about type safety across abi-encoding and -decoding. In situations where the data types are out of sync, and the target type cannot fit the input, the decoding call will revert. This will happen if, for example, upper bits of chainId are used to store data or the number of chains go beyond 2ˆ24 (unlikely).

Not settling on a single representation for the same thing can create confusion for future development. When doing cross-chain development, it is even more important to make sure everything is synchronized and well-understood across all the chains.

Recommendations

Consider using a canonical representation for all logical measurements and identifiers in the codebase. This includes items such as chainId, fees, gas, and other similar values. When using `abi.encode`, data is slotted into bytes32, which means that there is no advantage in using smaller data types unless they are packed. To ensure that cross-chain data encoding and decoding are working as expected, it is recommended to implement test cases specifically for this purpose.

Remediation

> ChainId is now consistently uint24 except for the mappings in the root port. This is done generally to make cross-chain calls cheaper, except in the CoreRootRouter which uses abi.encode for simplicity. (All BridgeAgents use abi.encodePacked to decrease message size)

This issue has been acknowledged by Maia DAO, and fixes were implemented in the following commits:

- 4a120be7
- 9330339a

3.12 Incorrect accounting of total and strategies debt

- **Target**: BranchPort
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The approved strategy contract is authorized to borrow assets via the `manage` function and repay debt via the `replenishReserves` function, but only up to the amount of `reservesLacking`.

The `_strategy` address, which will return debt, `amount` of tokens, and `_token` address, is controlled by the caller of the `replenishReserves` function.

The `reservesLacking` value shows the shortage of tokens needed to reach the minimum reserve amount. If `reservesLacking` is greater than the `_amount` value, then the `_amount` will be withdrawn; otherwise, the `reservesLacking` value will be withdrawn. Regardless of the withdrawal amount, both the `getPortStrategyTokenDebt` and `getStrategyTokenDebt` will be reduced by the `_amount` value.

Additionally, the value of the debt strategy will be reduced, not for the strategy that returned the funds but for the strategy that called the function.

```
function replenishReserves(address _strategy, address _token,
    uint256 _amount) external {
        if (!isStrategyToken[_token]) revert UnrecognizedStrategyToken();
        if (!isPortStrategy[_strategy][_token])
    revert UnrecognizedPortStrategy();

        uint256 reservesLacking = _reservesLacking(_token);

        IPortStrategy(_strategy).withdraw(address(this), _token, _amount
    < reservesLacking ? _amount : reservesLacking);

        getPortStrategyTokenDebt[msg.sender][_token] -= _amount;
        getStrategyTokenDebt[_token] -= _amount;

        emit DebtRepaid(_strategy, _token, _amount);
    }
```

Impact

The approved strategies contract is able to reduce debt by using funds withdrawn from other strategies. Additionally, there is an issue where if the value of `reservesLacking` is less than the amount of funds that need to be withdrawn, the debt counters will be reduced inaccurately. This can result in an incorrect calculation of the minimum number of reserves.

For example,

1. Current token balance is 100 tokens, the `getMinimumTokenReserveRatio[_token]` is 1000, the `_minimumReserves` is 10 tokens, and `_excessReserves` is 90 tokens.
2. The strategy is borrowed; all possible tokens amount 90 tokens.
3. The `getStrategyTokenDebt[_token] = 90` tokens.
4. The `currBalance` is equal to 10 tokens and `_minimumReserves` is still 10 tokens.
5. The `replenishReserves` function is called with `_amount` = 10 tokens.
6. Because the `_reservesLacking` is 0, the strategy will withdraw nothing, but `getStrategyTokenDebt[_token]` will reduced by `_amount`.
7. After changing the `getStrategyTokenDebt[_token]` without changing the balance, the `_minimumReserves` became equal to 9 tokens, but the `currBalance` is still equal to 10 tokens. So the `_excessReserves` will return 1 token, and strategies will be able to borrow again.

```
uint256 internal constant DIVISIONER = 1e4;

    function _excessReserves(address _token)
    internal view returns (uint256) {
        uint256 currBalance = ERC20(_token).balanceOf(address(this));
        uint256 minReserves = _minimumReserves(currBalance, _token);
        return currBalance > minReserves ? currBalance - minReserves : 0;
    }
    function _reservesLacking(address _token)
    internal view returns (uint256) {
        uint256 currBalance = ERC20(_token).balanceOf(address(this));
        uint256 minReserves = _minimumReserves(currBalance, _token);
        return currBalance < minReserves ? minReserves - currBalance : 0;
    }
    function _minimumReserves(uint256 _currBalance, address _token)
    internal view returns (uint256) {
        return ((_currBalance + getStrategyTokenDebt[_token])
    * getMinimumTokenReserveRatio[_token]) / DIVISIONER;
    }
```

Recommendations

We recommend implementing the function as shown below. Add a new `amount` variable equal to the actual number of withdrawn tokens and reduce `getPortStrategyTokenDebt` and `getStrategyTokenDebt` values by `amount`:

```
function replenishReserves(address _strategy, address _token,
    uint256 _amount) external {
        ...
    uint256 amount = _amount < reservesLacking ?
    _amount : reservesLacking;

    IPortStrategy(_strategy).withdraw(address(this), _token,
    _amount < reservesLacking ? _amount : reservesLacking);
    IPortStrategy(_strategy).withdraw(address(this), _token, amount);

    getPortStrategyTokenDebt[msg.sender][_token] -= _amount;
    getPortStrategyTokenDebt[_strategy][_token] -= amount;

    getStrategyTokenDebt[_token] -= _amount;
    getStrategyTokenDebt[_token] -= amount;

    emit DebtRepaid(_strategy, _token, _amount);
    emit DebtRepaid(_strategy, _token, amount);
}
```

Remediation

This issue has been acknowledged by Maia DAO, and fixes were implemented in the following commits:

- c11c18a1

- d04e441f

3.13 Bridging assets erroneously mints new assets

- **Target**: ArbitrumBranchPort
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The functions `bridgeIn` and `bridgeInMultiple` are supposed to increase the local `hToken` supply by bridging assets into the Arbitrum Branch. No assets are supposed to be minted here, only transferred from the RootPort. This can be done by calling `RootPort→bridgeToLocalBranch` with a deposit equal to 0. Instead, these functions both call `mintToLocalBranch`, which bridges nothing and mints new `hTokens` every time.

Impact

More tokens than expected will be minted, and the tokens will not leave the RootPort. This mistake can possibly be manually fixed by selectively burning. Depending on the exact use of the tokens, inflation of the number of tokens might be detrimental.

Recommendations

Replace calls to `mintToLocalBranch` with `bridgeToLocalBranch` instead and set the correct amount and deposits so that no new tokens are minted.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit 0fecbc05.

3.14 Lack of input validation

- **Target**: Multiple Contracts
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Low

Description

The following functions lack input validation.

1. In the BranchBridgeAgent contract,
   - The `anyFallback` function lacks a check that the `getDeposit` contains the `_depositNonce`.
2. In the UlyssesFactory contract,
   - The `createToken` function lacks a check that the `pools` contains the input `poolIds[i]`.
   - The `createPools` function lacks a check that `assets` contains zero addresses.
3. In the UlyssesPool contract,
   - The `swapIn` and `swapFromPool` functions lack a check that the `assets` is not zero.
4. In the UlyssesToken contract,
   - The `addAsset` function lacks a check that the `_weight` is not zero.
   - The `setWeights` function lacks a check that the `_weights` contains zero amounts.
5. In the ERC4626MultiToken contract,
   - The `constructor` lacks a check that the `_weights` contains zero amounts.
6. In the ERC20hTokenBranchFactory contract,
   - The `initialize` lacks a check that the `_coreRouter` is not zero address.
7. In the ERC20hTokenRootFactory contract,
   - The `constructor` lacks a check that the `rootPortAddress` is not zero address.
   - The `initialize` lacks a check that the `_coreRouter` is not zero address.
8. In the RootBridgeAgentFactory contract,
   - The `constructor` lacks a check that the `wrappedNativeToken`, `rootPortAddress`, and `daoAddress` are not zero addresses.

9. In the BranchBridgeAgentFactory contract,
    - The `createBridgeAgent` lacks a validation of `_rootBridgeAgentAddress`.
10. In the ERC20hTokenRoot contract,
    - The `constructor` lacks a check that the `factoryAddress` and `rootPortAddress` is not zero address.
11. In the BranchBridgeAgent contract,
    - The `constructor` lacks a check that the all passed addresses is not zero.
    - The `_clearDeposit` lacks a check that the `getDeposit[_depositNonce]` exists.
12. In the ArbitrumBranchPort contract,
    - The `constructor` lacks a check that the `rootPortAddress` address is not zero.
13. In the BranchPort contract,
    - The `initialize` lacks a check that the `coreBranchRouterAddress` and `_bridgeAgentFactory` addresses are not zero.
    - The `setCoreRouter` lacks a check that the `_newCoreRouter` address is not zero.
14. In the BaseBranchRouter contract,
    - The `initialize` lacks a check that the `localBridgeAgentAddress` address is not zero.
15. In the CoreBranchRouter contract,
    - The `constructor` lacks a check that the `localPortAddress` and `hTokenFactoryAddress` addresses are not zero.
16. In the BasePortGauge contract,
    - The `constructor` lacks a check that the `_bRouter` address is not zero.
17. In the CoreRootRouter contract,
    - The `constructor` lacks a check that the `_wrappedNativeToken` and `_rootPortAddress` addresses are not zero.
    - The `initialize` lacks a check that the `_bridgeAgentAddress` and `_hTokenFactory` addresses are not zero.
18. In the MulticallRootRouter contract,
    - The `constructor` lacks a check that the `_localPortAddress` and `_multicallAddress` addresses are not zero.

- The initialize lacks a check that the _bridgeAgentAddress address is not zero.

19. In the RootBridgeAgent contract,
    - The constructor lacks a check that all passed addresses are not zero.
    - The _reopenSettlement lacks a check that the getSettlement[_settlementNonce] exists.
    - The callOutAndBridge lacks a check that the IPort(localPortAddress).getLocalTokenFromGlobal() and the IPort(localPortAddress).getUnderlyingTokenFromLocal return nonzero addresses
    - The callOutAndBridgeMultiple lacks a check that the IPort(localPortAddress).getLocalTokenFromGlobal() and the IPort(localPortAddress).getUnderlyingTokenFromLocal return nonzero addresses.
    - The _bridgeIn lacks a check that the IPort(localPortAddress).getGlobalTokenFromLocal() returns nonzero addresses.
    - The _gasSwapIn and _gasSwapOut lack a check that the IPort(localPortAddress).getGasPoolInfo returns nonzero addresses.

20. In the RootPort contract,
    - The constructor lacks a check that the _wrappedNativeToken address is not zero.
    - The initialize lacks a check that the _bridgeAgentFactory and _coreRootRouter addresses are not zero.
    - The initializeCore lacks a check that the _coreLocalBranchBridgeAgent and _localBranchPortAddress addresses are not zero.
    - The forefeitOwnership lacks a check that the _owner address is not zero.
    - The setLocalBranchPort lacks a check that the _branchPort address is not zero.
    - The setUnderlyingAddress, setAddresses, setLocalAddress, addNewChain, initializeEcosystemTokenAddresses, and addEcosystemTokenToChain lack verification that adding token addresses does not lead to overwriting previously added ones.
    - The mint, burn, bridgeToRoot, bridgeToRootFromLocalBranch, bridgeToLocalBranch,and burnFromLocalBranch lack a check that the hToken address is actually created over ERC20hTokenRootFactory.

Impact

If important input parameters are not checked, especially in functions that are available for any user to call, it can result in functionality issues and unnecessary gas usage and can even be the root cause of critical problems. It is crucial to properly validate input parameters to ensure the correct execution of a function and prevent any unintended consequences.

Recommendations

Consider adding require statements and necessary checks to the above functions.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit 6ba3df02.

3.15 Lack of new owner address check

- **Target**: Multiple
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Informational

Description

The smart contracts UlyssesPool, UlyssesToken, BranchPort, BranchBridgeAgentFactory, and ERC20hTokenBranch inherit from the solady/auth/Ownable.sol contract and use the `_initializeOwner` function to assign an owner to the contract. The owner's address is provided by the caller within the `constructor`. However, there are no checks in place to ensure that the address of the new owner is not a zero address. Furthermore, the `_initializeOwner` function does not validate this either.

Impact

If a zero address is set as the `owner` during contract deployment, the contract will deploy successfully but the contract owner will not be set. This can potentially lead to an inability to perform certain functions, such as modifying the contract state.

Recommendations

We recommend implementing proper validation checks inside the `constructor` function before the `_initializeOwner` function execution in the UlyssesPool, UlyssesToken, BranchPort, BranchBridgeAgentFactory, and ERC20hTokenBranch contracts.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit da4751e6. Not fixed for the UlyssesToken contract.

3.16 Addresses may accidentally be overwritten

- **Target**: RootPort
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Informational

Description

The addEcosystemTokenToChain function may only be called by the contract owner. It adds entries to the double mappings getGlobalAddressFromLocal and getLocalAddressFromGlobal, which map between local and global address on a specific chain. However, there are no checks done to ensure that the mapping is not already set.

```
function addEcosystemTokenToChain(address ecoTokenGlobalAddress,
    address ecoTokenLocalAddress, uint256 toChainId)
    external
    onlyOwner
{
    getGlobalAddressFromLocal[toChainId][ecoTokenLocalAddress]
    = ecoTokenGlobalAddress;
    getLocalAddressFromGlobal[toChainId][ecoTokenGlobalAddress]
    = ecoTokenLocalAddress;
}
```

This can modify the addresses set by setAddresses, which is a function that can only be called by a coreRootRouterAddress. It also ends up replicating the behavior in a different function with a different modifier:

```
function setLocalAddress(address _globalAddress, address _localAddress,
    uint24 _fromChain)
    external
    requiresCoreBridgeAgent
{
    getGlobalAddressFromLocal[_fromChain][_localAddress]
    = _globalAddress;
    getLocalAddressFromGlobal[_fromChain][_globalAddress]
    = _localAddress;
}
```

Impact

The `addEcosystemTokenToChain` function can change addresses set by the core root router and vice versa. It is likely unintended that the owner can accidentally overwrite addresses that result from cross-chain communication.

Recommendations

If it is intentional that the owner should be able to override addresses set by the router, then consider renaming the `add` function to `set`. Otherwise, introduce a requirement that the address is not already set, possibly with a parameter that can override the behavior.

Remediation

This issue has been acknowledged by Maia DAO, and fixes were implemented in the following commits:

- 728ee138
- 8b17cb59

3.17 Ownable contracts allow renouncing

- **Target**: Multiple Contracts; e.g., BranchBridgeAgentFactory, RootPort, BranchPort, UlyssesToken, UlyssesPool
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Informational

Description

The `renounceOwnership()` function is included by default in contracts that inherit the Ownable contract. In some cases, this functionality is used intentionally, such as to initialize a contract and then permanently disable the possibility of initializing it again by renouncing ownership. However, in other contracts, the owner functionality is used throughout the contract for important functionality. If an owner accidentally renounces ownership, this *permanently* stops anyone from calling these critical functions. Therefore, it is important to use caution when using `renounceOwnership()` and to ensure that it is used only when necessary and intentional.

Impact

Depending on which contract becomes unusable, the impact could vary. It could result in the loss of funds or the loss of configurability of contracts, potentially requiring redeployment.

Recommendations

In contracts that require an active owner, override `renounceOwnership()` with a function that reverts. The Ownable contract contains a two-step ownership transfer procedure that can be used to change ownership in a secure way.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit ce37be6d.

> Note that UlyssesPool and UlyssesToken were not added because users or protocols may want to make the Pool or Token completely decentralized by “freezing” contract parameters.

3.18 Lack of validation in UlyssesFactory

- **Target**: UlyssesFactory
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Informational

Description

In `UlyssesFactory` → `createToken`, any caller can create a new `UlyssesToken` instance.

```
function createToken(uint256[] calldata poolIds, uint256[]
    calldata weights, address owner)
    external
    returns (uint256 _tokenId)
{
    _tokenId = ++tokenId;

    uint256 length = poolIds.length;
    address[] memory destinations = new address[](length);
    for (uint256 i = 0; i < length;) {
        destinations[i] = address(pools[poolIds[i]]);
        ...
    }
 ...
}
```

The function also takes in a list of `poolIds` to include as destinations for the token. The addresses for these are looked up in the `pools` mapping (`mapping(uint256 => UlyssesPool)`), but there is no verification that the ID of the pool was an actual address. If the ID is not found, the mapping will return 0, which makes the destination the zero address. The call will not fail until much later, when it tries to call the `decimals()` function on the address.

There is also no check to ensure there is at least one destination or that the `weights` array is the same length as `destinations`.

Impact

It is recommended to validate inputs explicitly and early, instead of relying on lucky reverts further down the chain. Zeroed data is the most common input in an accidental function call, and making sure that all relevant data is set can help prevent such errors. For instance, if a customer enters the wrong poolId, the resulting revert error will be that “`ERC20(address(0)).decimals()` is not a function”, which is not helpful and costs a lot of gas to discover.

Recommendations

Add explicit validation for input parameters and ensure that all mapping lookups return a nonzero value, as failing to do so may lead to unexpected behavior and errors that are hard to debug. In cases where a mapping lookup returns a zero value, it is likely that a mistake has been made, and it is best to catch this early on with proper validation.

Remediation

This issue has been acknowledged by Maia DAO, and a fix was implemented in commit ce1d3d1e.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Lack of validation for `_bridgeAgent`

The `addBridgeAgent` function enables the addition of a new trusted `_bridgeAgent` address along with its corresponding `manager` address. However, it is restricted to being called only by the `BridgeAgentFactory` contract that has been approved by the owner. Despite this restriction, there is no check within the function to ensure that the `_bridgeAgent` address being added has not already been added before, thus allowing for overwriting of its `_manager` address.

```
contract RootPort is Ownable, IRootPort {
     ...
    modifier requiresBridgeAgentFactory() {
        if (!isBridgeAgentFactory[msg.sender])
    revert UnrecognizedBridgeAgentFactory();
        _;
    }

    function toggleBridgeAgentFactory(address _bridgeAgentFactory)
    external onlyOwner {
        isBridgeAgentFactory[_bridgeAgentFactory]
    = !isBridgeAgentFactory[_bridgeAgentFactory];
    }

    function addBridgeAgent(address _manager, address _bridgeAgent)
    external requiresBridgeAgentFactory {
        bridgeAgents.push(_bridgeAgent);
        bridgeAgentsLenght++;
        getBridgeAgentManager[_bridgeAgent] = _manager;
        isBridgeAgent[_bridgeAgent] = !isBridgeAgent[_bridgeAgent];
    }
     ...
}
```

Although the current implementation of the RootBridgeAgentFactory contract does not permit the caller to provide the address of a previously created RootBridgeAgent contract, the RootPort owner is still able to add different implementations. To prevent the possibility of overwriting the `_manager` address, we recommend adding a check to ensure that the `_bridgeAgent` does not already exist within the `bridgeAgents` or that `getBridgeAgentManager[_bridgeAgent]` is not set. Additionally, it is advisable to validate that both the `_manager` and `_bridgeAgent` are not zero addresses.

4.2 Missing test coverage

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

This project has a few test cases that also utilize the property-based testing that Forge offers. This is a great start, but it would be advisable to expand the test coverage to include all contracts. Since the contracts employ use of multi-layered encoding and packing, as well as cross-chain function calls, end-to-end testing becomes even more crucial than usual.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects:

- Finds bugs and design flaws early (preaudit or prerelease).
- Gives insight into areas for optimization (e.g., gas cost).
- Displays code maturity.
- Bolsters customer trust in your product.
- Improves understanding of how the code functions, integrates and operates — for developers and auditors alike.
- Increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

Several of the findings in this audit would have been discovered with simple test cases. Negative test cases would have found missing permission checks, and positive test cases would have found both the functions that are not working and the functions where the side effects are unexpected. Because of the high amount of findings and the low test coverage, we expect there to be additional bugs yet to be discovered but which can be found by adding simple test cases.

4.3 No guards against reentrancy

Although we did not identify any attacks using reentrancy during this audit, the following functions lack protection against reentrancy:

- BranchPort: The `replenishReserves` function.
- MulticallRootRouter: The `anyExecuteSignedDepositMultiple`, the `anyExecuteSignedDepositSingle`, the `anyExecuteSigned`, and the `anyExecute` functions.
- RootBridgeAgent: The `callOutAndBridge`, the `callOutAndBridgeMultiple`, and the `anyExecute` functions.

We recommend implementing `lock` modifiers for functions that make external calls to user contracts to prevent reentrancy attacks.

4.4 Unhandled return

The `anyExecute` function in the RootBridgeAgent contract does not handle the status returned by the `IRouter(localRouterAddress)` contract function calls. This means that the `anyExecute` function always returns a `true` success status, and any errors or issues with the execution of external functions are ignored.

We recommend processing the status returned by `IRouter` contract function calls to ensure `anyExecute` accurately reflects the success or failure of external function calls.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: ArbitrumBranchBridgeAgent.sol

Function: `depositToPort(address underlyingAddress, uint256 amount)`

Allows any caller to deposit asset `amount` to `localPortAddress`. Function has a lock. Caller can provide an arbitrary `underlyingAddress` contract address which will be called.

Inputs

- `underlyingAddress`
  - **Validation**: no checks here, but corresponding globalToken address should exists inside the `IRootPort(rootPortAddress).getLocalTokenFromUnder(_underlyingAddress, localChainId)`.
  - **Impact**: the address of deposited token
- `amount`
  - **Validation**: no checks
  - **Impact**: the amount of tokens

Branches and code coverage (including function calls)

Intended branches

- the caller deposited the amount of `underlyingAddress` tokens
  - ☐ Test coverage
- `globalToken` associated with `underlyingAddress` will be minted for `msg.sender`
  - ☐ Test coverage

Negative behaviour

- revert if `underlyingAddress` token address is unknown
  - ☐ Negative test

- revert if caller doesn't have enough `underlyingAddress` tokens
  - ☐ Negative test

Function call analysis

- `IArbPort(localPortAddress).depositToPort(msg.sender, msg.sender, underlyingAddress, amount)`
  - **External/Internal?**: External
  - **Argument control?**: underlyingAddress, amount
  - **Impact**: check that the `globalToken` exists for `underlyingAddress` and mint `globalToken` `amount` tokens for the caller.

Function: `withdrawFromPort(address localAddress, uint256 amount)`

Allows withdraw asset tokens from the `localPortAddress` contract. Function has a lock.

Inputs

- `localAddress`
  - **Validation**: the `rootPort.getLocalAddressFromGlobal` array should contain this address as global token address
  - **Impact**: this tokens will be burned to exchange the `underlyingAddress` tokens
- `amount`
  - **Validation**: the caller should have more or equal amount of `localAddress` tokens
  - **Impact**: the amount of `localAddress` tokens will be burned and `underlyingAddress` will be received by the caller

Branches and code coverage (including function calls)

Intended branches

- the balance of `localAddress` of msg.sender was decreased by `amount`
  - ☐ Test coverage
- the caller receive the `amount` of `underlyingAddress` tokens
  - ☐ Test coverage

Negative behaviour

- `localAddress` token address is unknown
  - ☐ Negative test

- the caller doesn't have enough `localAddress` tokens
  - ☐ Negative test

Function call analysis

- `IArbPort(localPortAddress).withdrawFromPort(msg.sender, msg.sender, localAddress, amount) → IRootPort(rootPortAddress).getUnderlyingTokenFromLocal(_globalAddress, localChainId)`
  - **External/Internal?**: External
  - **Argument control?**: _globalAddress
  - **Impact**: return the `underlyingAddress` associated with `localAddress`. if `underlyingAddress` is zero, transaction will be reverted

5.2 Module: BranchBridgeAgent.sol

Function: `callOutAndBridgeMultiple(byte[] params, DepositMultipleInput dParams, uint128 rootExecutionGas)`

The same as the `callOutAndBridge` but user provides multiplies addresses of tokens for deposit and lock.

Function: `callOutAndBridge(byte[] params, DepositInput dParams, uint128 rootExecutionGas)`

Allows to perform a call to the Root Omnichain Router while depositing a single asset. The function has a lock. The arbitrary contracts controlled by user can be called. The new `Deposit` object will be saved inside the `getDeposit` with incremented nonce key. Function has a lock.

Inputs

- `params`
  - **Validation**: no checks
  - **Impact**: will encoded inside the data which is used for cross-chain call
- `dParams`
  - **Validation**: no checks
  - **Impact**: the data is contains the tokens addresses and amounts are used for `bridgeOutMultiple` call for depositing
- `rootExecutionGas`
  - **Validation**: no checks
  - **Impact**: will encoded inside the data which is used for cross-chain call. the

gas allocated for omnichain execution

Function call analysis

- `_callOutAndBridge(msg.sender, params, dParams, rootExecutionGas) → _depositAndCall(depositor, data, dParams.hToken, dParams.token, dParams.amount, dParams.deposit) → _createDepositSingle(_depositor, _hToken, _token, _amount, _deposit) → _createDepositMultiple(_user, hTokens, tokens, amounts, deposits) → IPort(localPortAddress).bridgeOutMultiple(_user, _hTokens, _tokens, _amounts, _deposits)`
  - **External/Internal?**: External
  - **Argument control?**: depositor, data, dParams.hToken, dParams.token, dParams.amount, dParams.deposit
  - **Impact**: deposit tokens into localPortAddress from msg.sender, the token address is provided as `_token` by caller. If the _amount > _deposit, than the difference between _amount and _deposit of `_hToken` will be deposited and burned, so the tokens will be locked
- `_callOutAndBridge(msg.sender, params, dParams, rootExecutionGas) → _depositAndCall(depositor, data, dParams.hToken, dParams.token, dParams.amount, dParams.deposit) → _createDepositSingle(_depositor, _hToken, _token, _amount, _deposit) → _createDepositMultiple(_user, hTokens, tokens, amounts, deposits) → wrappedNativeToken.deposit{value: msg.value}()`
  - **External/Internal?**: External
  - **Argument control?**: msg.value is controlled but should be more than `MIN_FALLBACK_OVERHEAD`
  - **Impact**: deposit to the wrapped native token contract the gas allocated for omnichain execution
- `_callOutAndBridge(msg.sender, params, dParams, rootExecutionGas) → _depositAndCall(depositor, data, dParams.hToken, dParams.token, dParams.amount, dParams.deposit) → _createDepositSingle(_depositor, _hToken, _token, _amount, _deposit) → _createDepositMultiple(_user, hTokens, tokens, amounts, deposits) → address(wrappedNativeToken).safeTransfer(localPortAddress, msg.value)`
  - **External/Internal?**: External
  - **Argument control?**: msg.value is controlled but should be more than `MIN_FALLBACK_OVERHEAD`
  - **Impact**: transfer this wrapped gas amount to the `localPortAddress`

Function: `callOutSignedAndBridgeMultiple(byte[] params, DepositMultipleInput dParams, uint128 rootExecutionGas)`

Deposit multiple tokens and send cross-chain request with data with type `bytes1(0x06)`.

Function: `callOutSignedAndBridge(byte[] params, DepositInput dParams, uint128 rootExecutionGas)`

Deposit tokens and send Cross-Chain request with data with type `bytes1(0x05)`.

Function: `callOutSigned(byte[] params, uint128 rootExecutionGas)`

The same as the `callOut` function, but inside the data will be encoded the `bytes1(0x04)` type.

Function: `callOut(byte[] params, uint128 rootExecutionGas)`

Allows to call the Root Omnichain Router without doing deposit, only gas amount will be deposited. The new `Deposit` object will be saved inside the `getDeposit` with incremented nonce key. Function has a lock.

1. Check msg.value
2. call `Port(localPortAddress).bridgeOutMultiple(_user, _hTokens, _tokens, _amounts, _deposits)`

Inputs

- `params`
  - **Validation**: no checks
  - **Impact**: will encoded inside the data which is used for cross-chain call
- `rootExecutionGas`
  - **Validation**: no checks
  - **Impact**: will encoded inside the data which is used for cross-chain call. the gas allocated for omnichain execution
- `msg.value`
  - **Validation**: `msg.value` should be more than `MIN_FALLBACK_OVERHEAD`
  - **Impact**: fallback gas amount, this value is deposited to the wrappedNativeToken and transferred to the localPortAddress

Function call analysis

- `_callOut(msg.sender, params, rootExecutionGas)` → `_depositAndCall(deposit`

`or, data, address(0), address(0), 0, 0) → _createDepositSingle(_depositor, _hToken, _token, _amount, _deposit) → _createDepositMultiple(_user, hTokens, tokens, amounts, deposits) → IPort(localPortAddress).bridgeOutMultiple(_user, _hTokens, _tokens, _amounts, _deposits)`
  - **External/Internal?**: External
  - **Argument control?**: _user = `msg.sender`, hToken = address(0), tokens[0] = address(0), amounts[0] = 0, deposits[0] = 0
  - **Impact**: the function will do nothing with zero amounts in this case
- `_callOut(msg.sender, params, rootExecutionGas) → _depositAndCall(depositor, data, address(0), address(0), 0, 0) → _createDepositSingle(_depositor, _hToken, _token, _amount, _deposit) → _createDepositMultiple(_user, hTokens, tokens, amounts, deposits) → wrappedNativeToken.deposit{value: msg.value}()`
  - **External/Internal?**: External
  - **Argument control?**: msg.value is controlled but should be more than `MIN_FALLBACK_OVERHEAD`
  - **Impact**: deposit to the wrapped native token contract the gas allocated for omnichain execution
- `_callOut(msg.sender, params, rootExecutionGas) → _depositAndCall(depositor, data, address(0), address(0), 0, 0) → _createDepositSingle(_depositor, _hToken, _token, _amount, _deposit) → _createDepositMultiple(_user, hTokens, tokens, amounts, deposits) → address(wrappedNativeToken).safeTransfer(localPortAddress, msg.value)`
  - **External/Internal?**: External
  - **Argument control?**: msg.value is controlled but should be more than `MIN_FALLBACK_OVERHEAD`
  - **Impact**: transfer this wrapped gas amount to the `localPortAddress`
- '_callOut(msg.sender, params, rootExecutionGas) -> _depositAndCall(depositor, data, address(0), address(0), 0, 0) -> _performCall(_data)-> IAnycallProxy(localAnyCallAddress).anyCal
rootBridgeAgentAddress, _calldata, rootChainId, AnycallFlags.FLAG_ALLOW_FALLBACK,
""

)'

- **External/Internal?**: External
  - **Argument control?**: _calldata
  - **Impact**: perform call to `localAnyCallAddress` contract with users `_calldata` for cross-chain messaging

Function: `redeemDeposit(uint32 _depositNonce)`

Allows to redeem tokens back to the owner of deposit. The function can be called by any caller, not only by the owner of deposit!!

Inputs

- `_depositNonce`
  - **Validation**: getDeposit[_depositNonce].status should not be equal Deposit-Status.Failed
  - **Impact**: id of an existing deposit

Branches and code coverage (including function calls)

Intended branches

- the `getDeposit[_depositNonce]` data is reset
  - ☐ Test coverage
- the owner of deposit received the deposited funds
  - ☐ Test coverage

Negative behaviour

- _depositNonce is invalid
  - ☐ Negative test
- msg.sender != owner of deposit
  - ☐ Negative test
- repeated redeem of funds without additional deposit is not possible
  - ☐ Negative test

Function call analysis

- `_redeemDeposit(_depositNonce) → IPort(localPortAddress).bridgeIn(deposit .owner, deposit.hTokens[i], deposit.amounts[i] - deposit.deposits[i])`
  - **External/Internal?**: External
  - **Argument control?**: Nothing
  - **Impact**: mint the value of the difference between `amounts` and `deposits` the `deposit.hTokens[i]` tokens to the `deposit.owner`
- `_redeemDeposit(_depositNonce) → IPort(localPortAddress).withdraw(deposit .owner, deposit.tokens[i], deposit.deposits[i])`
  - **External/Internal?**: External
  - **Argument control?**: Nothing
  - **Impact**: transfer the `deposits[i]` amount of the `tokens[i]` to the deposit.owner

- `_redeemDeposit(_depositNonce)` → `IPort(localPortAddress).withdraw(deposit.owner, address(wrappedNativeToken), deposit.depositedGas)`
  - **External/Internal?**: External
  - **Argument control?**: Nothing
  - **Impact**: transfer the `deposit.depositedGas` amount of the `wrappedNativeToken` to the deposit.owner

Function: `retrySettlement(uint32 _settlementNonce)`

Send cross-chain request with data with type `bytes1(0x07)` without depositing the tokens.

5.3 Module: ERC4626MultiToken.sol

Function: `deposit(uint256[] assetsAmounts, address receiver)`

Allows deposit assets. The caller provide the array with corresponding amount of asset tokens. There is no `assetsAmounts` length check, so callers can provide not all assets tokens.

Inputs

- `assetsAmounts`
  - **Validation**: no checks.
  - **Impact**: contains the number of tokens that the caller will provide in exchange for shares.
- `receiver`
  - **Validation**: there is a check that != address(0) inside the `_mint` function
  - **Impact**: the owner of minted shares

Branches and code coverage (including function calls)

Intended branches

- assetsAmounts.length == assets.length
  - [ ] Test coverage
- the caller provided assets tokens
  - [ ] Test coverage
- the receiver owns expected amount of shares
  - [ ] Test coverage

Negative behaviour

- assetsAmounts.length < assets.length
  - ☐ Negative test
- assetsAmounts.length > assets.length
  - ☐ Negative test
- receiver is zero
  - ☐ Negative test
- assetsAmounts contains zero amounts This issue has been acknowledged by Maia DAO.
  - ☐ Negative test

Function call analysis

- `previewDeposit(assetsAmounts)`
  - **External/Internal?**: Internal
  - **Argument control?**: assetsAmounts
  - **Impact**: calculate the minimum amount of shares for a given assetsAmounts.
- `receiveAssets(assetsAmounts) → assets[i].safeTransferFrom(msg.sender, address(this), assetsAmounts[i]);`
  - **External/Internal?**: External
  - **Argument control?**: assetsAmounts
  - **Impact**: transfer the number of tokens provided by the caller according to the addresses specified by the contract owner.
- `_mint(receiver, shares)`
  - **External/Internal?**: Internal
  - **Argument control?**: receiver
  - **Impact**: mint the `shares` amount of tokens for the receiver

Function: `mint(uint256 shares, address receiver)`

Allows mint exactly shares to receiver by depositing assets. The caller will provide the corresponding amount of each asset token.

Inputs

- `shares`
  - **Validation**: the caller should have enough amount of asset tokens to get the accordingly amount of shares
  - **Impact**: the expected number of shares to be minted
- `receiver`
  - **Validation**: there is a check that != address(0) inside the `_mint` function
  - **Impact**: the owner of minted shares

Branches and code coverage (including function calls)

Intended branches

- the expected number of assets of each asset token was transferred from the caller
  - ☐ Test coverage
- the balance of receiver increase by shares number
  - ☐ Test coverage

Negative behaviour

- caller doesn't have enough tokens
  - ☐ Negative test

Function call analysis

- `previewMint(shares)`
  - **External/Internal?**: Internal
  - **Argument control?**: `shares`
  - **Impact**: returns the number of assets rounded up to be transferred from the caller to mint the exact shares
- `receiveAssets(assetsAmounts)` → `assets[i].safeTransferFrom(msg.sender, address(this), assetsAmounts[i]);`
  - **External/Internal?**: External
  - **Argument control?**: nothing
  - **Impact**: transfer each ERC20 `asset` tokens from the caller to contract. the `assetsAmounts` calculated inside the `previewMint`
- `_mint(receiver, shares);`
  - **External/Internal?**: Internal
  - **Argument control?**: `receiver`
  - **Impact**: mint the `shares` amount of tokens for the receiver

Function: `redeem(uint256 shares, address receiver, address owner)`

Allows redeems a specific number of shares from owner and send each asset tokens to receiver

Inputs

- `shares`
  - **Validation**: allowance[owner][msg.sender] >= shares and balance of owner should be more or equal to shares

    - **Impact**: the exact number of shares to be burned
- `receiver`
    - **Validation**: there is a check is not zero address inside the `_transfer` function
    - **Impact**: the receiver of assets tokens
- `owner`
    - **Validation**: if msg.sender is not an owner, the msg.sender should have an approve from owner. owner should have enough shares
    - **Impact**: the owner of shares which will be burned.

Branches and code coverage (including function calls)

Intended branches

- the balance of owner decrease by `shares`
    - ☐ Test coverage
- the receiver have got `assets` number of tokens of each asset tokens.
    - ☐ Test coverage
- the `allowance` was decreased by `shares`
    - ☐ Test coverage

Negative behaviour

- msg.sender is not an owner and doesn't have an approve
    - ☐ Negative test
- approve isn't enough
    - ☐ Negative test
- the owner balance of shares less than `shares`
    - ☐ Negative test
- receiver is zero address
    - ☐ Negative test

Function call analysis

- `previewRedeem(shares)`
    - **External/Internal?**: Internal
    - **Argument control?**: shares
    - **Impact**: returns the amounts of assets rounded down to be transferred to the recipient.
- `_burn(owner, shares);`
    - **External/Internal?**: Internal
    - **Argument control?**: owner
    - **Impact**: burn shares from owner

- `sendAssets(assetsAmounts)` → `assets[i].safeTransfer(address(this), assetsAmounts[i]);`
  - **External/Internal?**: External
  - **Argument control?**: assetsAmounts
  - **Impact**: transfer each asset tokens to the receiver. The assetsAmounts is calculated inside the `previewRedeem`

Function: `withdraw(uint256[] assetsAmounts, address receiver, address owner)`

Allows burn shares from the owner and sends exactly the number of each asset token to the receiver. There is no `assetsAmounts` length check, so callers can receive not all assets tokens.

Inputs

- `assetsAmounts`
  - **Validation**: no checks
  - **Impact**: the exact number of each assets tokens will transfer to the receiver
- `receiver`
  - **Validation**: there is a check is not zero address inside the `_transfer` function
  - **Impact**: the receiver of assets tokens
- `owner`
  - **Validation**: if msg.sender is not an owner, the msg.sender should have an approve from owner. owner should have enough shares
  - **Impact**: the owner of shares which will be burned.

Branches and code coverage (including function calls)

Intended branches

- the balance of owner decrease by `shares`
  - ☐ Test coverage
- the receiver have got `assets` tokens of each `asset`
  - ☐ Test coverage
- the `allowance` was decreased by `shares`
  - ☐ Test coverage

Negative behaviour

- msg.sender is not an owner and doesn't have an approve
  - ☐ Negative test
- approve isn't enough

  - ☐ Negative test
- the owner balance of shares less than `shares`
  - ☐ Negative test
- receiver is zero address
  - ☐ Negative test

Function call analysis

- `previewWithdraw(assets)`
  - **External/Internal?**: Internal
  - **Argument control?**: assets
  - **Impact**: returns the maximum number of shares rounded up
- `_burn(owner, shares);`
  - **External/Internal?**: Internal
  - **Argument control?**: owner
  - **Impact**: burn shares from owner
- `sendAssets(assetsAmounts)` → `assets[i].safeTransfer(address(this), assetsAmounts[i]);`
  - **External/Internal?**: External
  - **Argument control?**: assetsAmounts
  - **Impact**: transfer each asset tokens to the receiver

5.4 Module: ERC4626.sol

Function: `deposit(uint256 assets, address receiver)`

Allows deposit assets of underlying tokens. Instead, the receiver will own the minted shares.

Inputs

- `assets`
  - **Validation**: if the caller cannot transfer `assets` amount of tokens, transaction will be reverted
  - **Impact**:
- `receiver`
  - **Validation**: there is a check that != address(0) inside the `_mint` function
  - **Impact**: the owner of minted shares

Branches and code coverage (including function calls)

Intended branches

- the balance of receiver increase by shares number
  - ☐ Test coverage
- the caller transfer the `assets` amount of asset token
  - ☐ Test coverage

Negative behaviour

- caller doesn't have enough `asset` tokens
  - ☐ Negative test

Function call analysis

- `previewDeposit(assets)`
  - **External/Internal?**: Internal
  - **Argument control?**: `assets`
  - **Impact**: returns the number of shares rounded down to be minted
- `address(asset).safeTransferFrom(msg.sender, address(this), assets);`
  - **External/Internal?**: External
  - **Argument control?**: `assets`
  - **Impact**: transfer ERC20 `asset` tokens from the caller to contract.
- `_mint(receiver, shares);`
  - **External/Internal?**: Internal
  - **Argument control?**: `receiver`
  - **Impact**: mint the `shares` amount of tokens for the receiver

Function: `mint(uint256 shares, address receiver)`

Allows mint exactly shares to receiver by depositing assets.

Inputs

- `shares`
  - **Validation**: the caller should have enough amount of asset tokens to get the accordingly amount of shares
  - **Impact**: the expected number of shares to be minted
- `receiver`
  - **Validation**: there is a check that != address(0) inside the `_mint` function
  - **Impact**: the owner of minted shares

Branches and code coverage (including function calls)

Intended branches

- the expected number of assets was transferred from the caller
  - ☐ Test coverage
- the balance of receiver increase by shares number
  - ☐ Test coverage

Negative behaviour

- caller doesn't have enough `asset` tokens
  - ☐ Negative test

Function call analysis

- `previewMint(shares)`
  - **External/Internal?**: Internal
  - **Argument control?**: `shares`
  - **Impact**: returns the number of assets rounded up to be transferred from the caller to mint the exact shares
- `address(asset).safeTransferFrom(msg.sender, address(this), assets);`
  - **External/Internal?**: External
  - **Argument control?**: `assets`
  - **Impact**: transfer ERC20 `asset` tokens from the caller to contract.
- `_mint(receiver, shares);`
  - **External/Internal?**: Internal
  - **Argument control?**: `receiver`
  - **Impact**: mint the `shares` amount of tokens for the receiver

Function: `redeem(uint256 shares, address receiver, address owner)`

Allows redeems a specific number of shares from owner and send asset tokens to receiver

Inputs

- `shares`
  - **Validation**: allowance[owner][msg.sender] >= shares and balance of owner should be more or equal to shares
  - **Impact**: the exact number of shares to be burned
- `receiver`
  - **Validation**: there is a check is not zero address inside the `_transfer` function

    - **Impact**: the receiver of assets tokens
- `owner`
    - **Validation**: if msg.sender is not an owner, the msg.sender should have an approve from owner. owner should have enough shares
    - **Impact**: the owner of shares which will be burned.

Branches and code coverage (including function calls)

Intended branches

- the balance of owner decrease by `shares`
    - ☐ Test coverage
- the receiver have got `assets` number of tokens
    - ☐ Test coverage
- the `allowance` was decreased by `shares`
    - ☐ Test coverage

Negative behaviour

- msg.sender is not an owner and doesn't have an approve
    - ☐ Negative test
- approve isn't enough
    - ☐ Negative test
- the owner balance of shares less than `shares`
    - ☐ Negative test
- receiver is zero address
    - ☐ Negative test

Function call analysis

- `previewRedeem(shares)`
    - **External/Internal?**: Internal
    - **Argument control?**: shares
    - **Impact**: returns the amount of assets rounded down to be transferred to the recipient.
- `_burn(owner, shares);`
    - **External/Internal?**: Internal
    - **Argument control?**: owner
    - **Impact**: burn shares from owner
- `address(asset).safeTransfer(receiver, assets);`
    - **External/Internal?**: External
    - **Argument control?**: assets

  - **Impact**: transfer asset tokens to the receiver

Function: `withdraw(uint256 assets, address receiver, address owner)`

Allows burn shares from the owner and sends exactly the number of asset tokens to the receiver.

Inputs

- `assets`
  - **Validation**: there is an indirect check that the owner must have the appropriate this number of assets number of shares
  - **Impact**: the exact number of asset tokens will transfer to the receiver
- `receiver`
  - **Validation**: there is a check is not zero address inside the `_transfer` function
  - **Impact**: the receiver of assets tokens
- `owner`
  - **Validation**: if msg.sender is not an owner, the msg.sender should have an approve from owner. owner should have enough shares
  - **Impact**: the owner of shares which will be burned.

Branches and code coverage (including function calls)

Intended branches

- the balance of owner decrease by `shares`
  - ☐ Test coverage
- the receiver have got `assets` number of tokens
  - ☐ Test coverage
- the `allowance` was decreased by `shares`
  - ☐ Test coverage

Negative behaviour

- msg.sender is not an owner and doesn't have an approve
  - ☐ Negative test
- approve isn't enough
  - ☐ Negative test
- the owner balance of shares less than `shares`
  - ☐ Negative test
- receiver is zero address
  - ☐ Negative test

Function call analysis

- `previewWithdraw(assets)`
  - **External/Internal?**: Internal
  - **Argument control?**: assets
  - **Impact**: returns the corresponding number of shares rounded up
- `_burn(owner, shares);`
  - **External/Internal?**: Internal
  - **Argument control?**: owner
  - **Impact**: burn shares from owner
- `address(asset).safeTransfer(receiver, assets);`
  - **External/Internal?**: External
  - **Argument control?**: assets
  - **Impact**: transfer asset tokens to the receiver

5.5 Module: MulticallRootRouter.sol

Function: `initialize(address _bridgeAgentAddress)`

The function can only be called once. Allows owner to initialize the `bridgeAgentAddress` address and reset ownership.

5.6 Module: RootPort.sol

Function: `addBridgeAgent(address _manager, address _bridgeAgent)`

Allows `BridgeAgentFactory` to add new `_bridgeAgent` address and manager address. Factory deploys the `_bridgeAgent` contract, the manager is address who initiate the `createBridgeAgent` call.

Inputs

- `_manager`
  - **Validation**: no checks
  - **Impact**: the caller of `createBridgeAgent` function. This address will be able to call function of `_bridgeAgent` available only for manager. But the `_bridgeAgent` doesn't keep this address, so check it over this contract (`getBridgeAgentManager`).
- `_bridgeAgent`
  - **Validation**: no checks

  - **Impact**: the address of new `RootBridgeAgent` contract. For BridgeAgents contracts available `requiresBridgeAgent` functions.

Branches and code coverage (including function calls)

Intended branches

- new `_bridgeAgent` added
  - ☐ Test coverage
- _manager is manager of `_bridgeAgent`
  - ☐ Test coverage

Negative behaviour

- _bridgeAgent already added
  - ☐ Negative test
- caller is not `requiresBridgeAgentFactory`
  - ☐ Negative test

Function: `addEcosystemTokenToChain(address ecoTokenGlobalAddress, address ecoTokenLocalAddress, uint256 toChainId)`

Allows owner of contract add addresses to `getGlobalAddressFromLocal` and `getLocalAddressFromGlobal` mapping. There aren't any checks so it is possible to rewrite existing addresses

Function: `initializeEcosystemTokenAddresses(address hermesGlobalAddress, address maiaGlobalAddress)`

Allows owner of contract add `hermes` and `maia` global addresses to `getGlobalAddressFromLocal` and `getLocalAddressFromGlobal` mapping. There aren't any checks so it is possible to rewrite existing addresses

Function: `syncBranchBridgeAgentWithRoot(address _newBranchBridgeAgent, address _rootBridgeAgent, uint24 _branchChainId)`

only for requiresCoreBridgeAgent. Allows coreBridgeAgent to set the `_newBranchBridgeAgent` address for `_rootBridgeAgent` contract.

Inputs

- `_newBranchBridgeAgent`
  - **Validation**: no checks
  - **Impact**: the address of BranchBridgeAgent

- _rootBridgeAgent
  - **Validation**: IBridgeAgent(_rootBridgeAgent).getBranchBridgeAgent(_branchChainId) should be zero and _newBranchBridgeAgent should be allowed by manager of _rootBridgeAgent
  - **Impact**: the address of contract _rootBridgeAgent which will be called to set _newBranchBridgeAgent
- _branchChainId
  - **Validation**: IBridgeAgent(_rootBridgeAgent).getBranchBridgeAgent(_branchChainId) should be zero
  - **Impact**: chainId of the chain to set the _newBranchBridgeAgent for

Branches and code coverage (including function calls)

Intended branches

- getBranchBridgeAgent for _branchChainId is set to _newBranchBridgeAgent value inside the _rootBridgeAgent contract
  - ☐ Test coverage

Negative behaviour

- caller is not requiresCoreBridgeAgent
  - ☐ Negative test
- getBranchBridgeAgent already set
  - ☐ Negative test

Function call analysis

- IBridgeAgent(_rootBridgeAgent).getBranchBridgeAgent(_branchChainId)
  - **External/Internal?**: External
  - **Argument control?**: _branchChainId
  - **Impact**: return the current BranchBridgeAgent contract address
- IBridgeAgent(_rootBridgeAgent).isBranchBridgeAgentAllowed(_branchChainId, _newBranchBridgeAgent)
  - **External/Internal?**: External
  - **Argument control?**: _branchChainId, _newBranchBridgeAgent
  - **Impact**: return true, if _newBranchBridgeAgent is allowed by manager of _rootBridgeAgent
- IBridgeAgent(_rootBridgeAgent).syncBranchBridgeAgent(_newBranchBridgeAgent, _branchChainId)
  - **External/Internal?**: External
  - **Argument control?**:

- _branchChainId, _newBranchBridgeAgent
  - **Impact**: set `_newBranchBridgeAgent` address for `_rootBridgeAgent` contract

Function: `toggleBridgeAgent(address _bridgeAgent)`

Allows owner of contract activate/deactivate BridgeAgent address.

5.7 Module: UlyssesFactory.sol

Function: `createPool(ERC20 asset, address owner)`

Allows anyone to create the new pool contract with arbitrary `asset` and `owner`.

Inputs

- `asset`
  - **Validation**: no checks
  - **Impact**: the underlying asset token that is used for UlyssesERC2426 initialization
- `owner`
  - **Validation**: no checks
  - **Impact**: owner of pool

Branches and code coverage (including function calls)

Intended branches

- new pool created properly
  ☐ Test coverage

Negative behaviour

- zero asset address
  ☐ Negative test
- zero owner address
  ☐ Negative test

Function call analysis

- `_createPool(ERC20 asset, address owner)` → `UlyssesPoolDeployer.deployPool(_poolId, address(asset), "Ulysses Pool", "ULP", owner, address(this));`
  - **External/Internal?**: Internal (library)
  - **Argument control?**: asset, owner

  - **Impact**: deploy a new Ulysses pool

Function: `createToken(uint256[] poolIds, uint256[] weights, address owner)`

Allows any caller to deploy the `UlyssesToken` contract using the existed `pools` addresses as `_assets` and arbitrary weights.

Inputs

- `poolIds`
  - **Validation**: no checks
  - **Impact**: the id's of pools created over `_createPool`
- `weights`
  - **Validation**: no checks
  - **Impact**: weights for the corresponding pools
- `owner`
  - **Validation**: no checks
  - **Impact**: the owner of new UlyssesToken

Branches and code coverage (including function calls)

Intended branches

- owner owns new UlyssesToken address
  - ☐ Test coverage

Negative behaviour

- `poolIds` contains non existed pool ids
  - ☐ Negative test
- `weights` contains zero values
  - ☐ Negative test
- `owner` is zero address
  - ☐ Negative test
- the poolIds.length != weights.length
  - ☐ Negative test

Function call analysis

- `new UlyssesToken(_tokenId,destinations,weights,"Ulysses Token","ULT",owner);`
  - **External/Internal?**: External

- **Argument control?**: destinations, weights, owner
- **Impact**: deploy new UlyssesToken contract with arbitrary weights and owner address, the caller can select any of the existing pools created over createPool or createPools functions.

5.8 Module: UlyssesPool.sol

Function: `claimProtocolFees()`

Calculate the amount of tokens which can be redeemed by owner of contract (protocol fee).

Branches and code coverage (including function calls)

Intended branches

- the expected amount of tokens was transferred to the owner
  - ☐ Test coverage

Negative behaviour

- caller is not an owner
  - ☐ Negative test

Function call analysis

- `asset.safeTransfer(factory.owner(), claimed)`
  - **External/Internal?**: External
  - **Argument control?**: nothing
  - **Impact**: transfer of the full available fee to the owner of the factory contract

5.9 Module: UlyssesToken.sol

Function: `addAsset(address asset, uint256 _weight)`

Allows owner of contract add new asset address and corresponding weight.

Inputs

- `asset`
  - **Validation**: ERC20(asset).decimals() == 18
  - **Impact**: new asset address

- `_weight`
  - **Validation**: no checks.
  - **Impact**: new asset weight.

Branches and code coverage (including function calls)

Intended branches

- `totalWeights` was increased by `_weight`
  - ☐ Test coverage
- `assets` contains new `asset`
  - ☐ Test coverage
- the caller provide proper amount of `asset` tokens to contract
  - ☐ Test coverage
- the caller receive the surplus tokens after updating balances
  - ☐ Test coverage

Negative behaviour

- `asset` is zero address
  - ☐ Negative test
- `_weight` is zero
  - ☐ Negative test

Function call analysis

- `updateAssetBalances → assets[i].balanceOf(address(this))`
  - **External/Internal?**: External
  - **Argument control?**: n/a
  - **Impact**: the current assets balance. used to calculate the balance changing.
- `updateAssetBalances → assets[i].safeTransfer(msg.sender, assetBalance - newAssetBalance);`
  - **External/Internal?**: External
  - **Argument control?**: n/a
  - **Impact**: transfer the surplus tokens to the caller
- `updateAssetBalances → assets[i].safeTransferFrom(msg.sender, address(this), newAssetBalance - assetBalance);`
  - **External/Internal?**: External
  - **Argument control?**: n/a
  - **Impact**: transfer the required amount of tokens to the contract from the caller

Function: `removeAsset(address asset)`

Allows owner of contract remove existed `asset`. The `asset` balance will transferred to the caller, also the caller will provide necessary amount of other assets.

Inputs

- `asset`
  - **Validation**: If `asset` isn't exist function will revert,
  - **Impact**: the `asset` address which will be removed from `assets` list, also there will be removed corresponding `weights` and index from `assetId`. The `assets` contains the addresses of tokens available for deposit into the Vault. Only the owner can change this list.

Branches and code coverage (including function calls)

Intended branches

- `asset` address was removed from `assets`
  - ☐ Test coverage
- corresponding weight was removed from `weights`
  - ☐ Test coverage
- the caller provide required tokens for each asset token address
  - ☐ Test coverage
- the caller receive the full `asset` token balance of contract
  - ☐ Test coverage
- `totalWeights` was decreased by corresponding weight
  - ☐ Test coverage

Negative behaviour

- `asset` does not exist
  - ☐ Negative test
- `asset` the last assets.
  - ☐ Negative test

Function call analysis

- `updateAssetBalances → assets[i].balanceOf(address(this))`
  - **External/Internal?**: External
  - **Argument control?**: n/a
  - **Impact**: the current assets balance. used to calculate the balance changing.
- `updateAssetBalances → assets[i].safeTransfer(msg.sender, assetBalance - n`

`ewAssetBalance);`
  - **External/Internal?**: External
  - **Argument control?**: n/a
  - **Impact**: transfer the surplus tokens to the caller
- `updateAssetBalances → assets[i].safeTransferFrom(msg.sender, address(this), newAssetBalance - assetBalance);`
  - **External/Internal?**: External
  - **Argument control?**: n/a
  - **Impact**: transfer the required amount of tokens to the contract from the caller
- `asset.safeTransfer(msg.sender, asset.balanceOf(address(this)));`
  - **External/Internal?**: External
  - **Argument control?**: asset
  - **Impact**: transfer full asset balance of contract to caller

Function: `setWeights(uint256[] _weights)`

Allows owner of contract update all weights. the balances will be recalculated

Inputs

- `_weights`
  - **Validation**: _weights.length != assets.length. but there isn't check that weights in non zero.
  - **Impact**: the new weights of the assets

Branches and code coverage (including function calls)

Intended branches

- the caller provided enough amount of tokens or received the surplus tokens
  - ☐ Test coverage

Negative behaviour

- new weight is zero
  - ☐ Negative test

Function call analysis

- `updateAssetBalances → assets[i].balanceOf(address(this))`
  - **External/Internal?**: External
  - **Argument control?**: n/a

    - **Impact**: the current assets balance. used to calculate the balance changing.
- `updateAssetBalances → assets[i].safeTransfer(msg.sender, assetBalance - newAssetBalance);`
    - **External/Internal?**: External
    - **Argument control?**: n/a
    - **Impact**: transfer the surplus tokens to the caller
- `updateAssetBalances → assets[i].safeTransferFrom(msg.sender, address(this), newAssetBalance - assetBalance);`
    - **External/Internal?**: External
    - **Argument control?**: n/a
    - **Impact**: transfer the required amount of tokens to the contract from the caller

6 Audit Results

At the time of our audit, the code was not deployed to mainnet EVM.

During our audit, we discovered 18 findings. Of these, five were critical risk, five were high risk, three were medium risk, one was low risk, and four were suggestions (informational). Maia DAO acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.