Zellic

Prepared for
Maia DAO
Maia DAO

Prepared by
Katerina Belotskaya
Ulrich Myhre
Zellic

December 21, 2023

Maia DAO Ulysses Protocol
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana, as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional infosec and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Executive Summary

Zellic conducted a security assessment for Maia DAO from November 22nd to December 12th, 2023. During this engagement, Zellic reviewed Maia DAO 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:

- Could an attacker perform a reentrancy attack on the Ports or Bridge Agent contracts?
- Can a distributed denial-of-service (DDOS) attack be conducted against the Bridge Agent contracts?
- Could users other than the owner access the Virtual Account contract?
- Are there any issues with the retry-, retrieve-, and redeem-deposit processes and with settlement state management?

1.2. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody

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

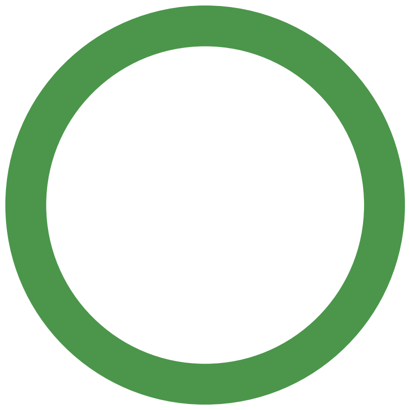
1.3. Results

During our assessment on the scoped Maia DAO Ulysses Protocol contracts, we discovered two findings, both of which were low impact.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	2
⬜ Informational	0



2. Introduction

2.1. About Maia DAO Ulysses Protocol

Maia DAO 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 other 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 accross 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 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 codebase 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, upgradability 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Maia DAO Ulysses Protocol Contracts

Repository	https://github.com/Maia-DAO/ulysses-layer-zero ↗
Version	ulysses-layer-zero: `35bee3b6686ab3616ebb835c6728593af978a7fd`
Programs	• BranchBridgeAgent • BranchBridgeAgentExecutor • BranchPort • RootBridgeAgent • RootBridgeAgentExecutor • RootPort.sol • VirtualAccount • AddressCodeSize • DecodeBridgeInMultipleParams
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 five 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:

Katerina Belotskaya
Engineer
kate@zellic.io

Ulrich Myhre
Engineer
unblvr@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 21, 2023	Start of primary review period
November 24, 2023	Kick-off call
December 20, 2023	End of primary review period

3. Detailed Findings

3.1. Lack of input validation

Target	Multiple Contracts		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The audit has identified several functions across various contracts that currently lack essential input-validation checks. Implementing these checks can significantly improve the robustness and security of the protocol. Below is the list of functions:

- The `sweep` function of the RootPort contract lacks a check that `_recipient` is not zero address.
- The `retrySettlement` function of the BranchBridgeAgent contract lacks a check that `executionState[_settlementNonce]` is equal to `STATUS_READY`. Performing this check can save gas and reduce commission fees by reverting invalid requests early in the `retrySettlement` function rather than later in the `lzReceiveNonBlocking` function after completing the transferring of two cross-chain messages.
- The `replenishReserves(address _token, uint256 _amount)` function of the BranchPort contract lacks a check that `_amount` and `getPortStrategyTokenDebt[msg.sender][_token]` are not zero values. This will prevent futile transactions and calls of arbitrary user's contracts.
- The `replenishReserves(address _strategy, address _token)` function of the BranchPort contract lacks a check that `getPortStrategyTokenDebt[_strategy][_token]` is not zero value. This will prevent futile transactions and calls of arbitrary user's contracts.
- The `callOutAndBridge` and `callOutAndBridgeMultiple` functions of the RootBridgeAgent contract lacks a check that the `_settlementOwnerAndGasRefundee` address is not zero.
- The `lzReceiveNonBlocking` function of the RootBridgeAgent contract for `retrySettlement` case does not verify that `settlement.status` is not `STATUS_FAILED`. This check will happen only after the `IRouter(rootRouterAddress).executeRetrySettlement` call, which triggers `IBridgeAgent(bridgeAgentAddress).retrySettlement`. But it is better to perform critical checks as early as possible in the call stack.
- The `redeemSettlement` function in the RootBridgeAgent contract verifies that the settlement status is not `STATUS_SUCCESS`. However, a more precise approach would be to check that the status is `STATUS_FAILED`, as this indicates the settlement is ready for the redeem action.

- The `bridgeIn` and `bridgeInMultiple` of the BranchPort contract lack a `lock` modifier against reentrancy attacks.
- The `bridgeToRoot`, `bridgeToBranch`, `bridgeToRootFromLocalBranch`, `bridgeToLocalBranchFromRoot`, `burnFromLocalBranch`, and `mintToLocalBranch` functions of the RootPort contract lack a `lock` modifier against reentrancy attacks.

Impact

If important input parameters are not checked, 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 to prevent 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 `bf16debc` ↗.

3.2. Reentrancy in the manage function

Target	BranchPort		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `manage` function permits trusted strategy contracts to borrow funds from the BranchPort contract. However, a strategy contract is restricted from withdrawing more token funds than its reserve management limit. Therefore, the total of funds already borrowed plus the new amount must not exceed this limit.

```
function _enforceReservesLimit(
    address _token,
    uint256 _amount,
    uint256 _strategyTokenDebt,
    uint256 _portStrategyTokenDebt
) internal view {
    ...
    if ((_amount + _portStrategyTokenDebt) >
    _strategyReserveManagementLimit(_token, totalTokenBalance)) {
        revert ExceedsReserveRatioManagementLimit();
    }
}

function _strategyReserveManagementLimit(address _token,
    uint256 _totalTokenBalance)
    internal
    view
    returns (uint256)
{
    return
        (_totalTokenBalance * (DIVISIONER
    - strategyReserveRatioManagementLimit[msg.sender][_token]))
    / DIVISIONER;
}
```

The `replenishReserves` function enables the return of funds from the strategy back to the BranchPort contract. Additionally, the `getPortStrategyTokenDebt` and `getStrategyTokenDebt` amounts are reduced prior to the external `withdraw` call. Since the `manage` function lacks a `lock` modifier to prevent reentrancy attacks, a strategy contract may reenter the `manage` function, potentially increasing the amount of funds it can borrow.

```
function replenishReserves(address _token, uint256 _amount)
    external override lock {
    getPortStrategyTokenDebt[msg.sender][_token] -= _amount;
    getStrategyTokenDebt[_token] -= _amount;

    uint256 currBalance = ERC20(_token).balanceOf(address(this));

    IPortStrategy(msg.sender).withdraw(address(this), _token, _amount);

    require(ERC20(_token).balanceOf(address(this)) - currBalance ==
    _amount, "Port Strategy Withdraw Failed");
    emit DebtRepaid(msg.sender, _token, _amount);
}
```

Impact

After an external call is executed, the `replenishReserves` function verifies the balance, requiring the strategy to return any additional borrowed funds by the end of the transaction. Consequently, the impact is not critical, as only trusted strategy contracts have the capability to invoke the `manage` function.

Recommendations

Given that the potential for an attack remains a threat to the contract's security, we advise implementing reentrancy attack safeguards within the `manage` function. Additionally, it is recommended to reduce the amounts of `getPortStrategyTokenDebt` and `getStrategyTokenDebt` subsequent to the external call.

Remediation

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

- e8f0a112↗
- 0b784036↗

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Insufficient verification in multiple CoreRootRouter contract functions

The `toggleStrategyToken`, `updateStrategyToken`, `togglePortStrategy`, and `updatePort-Strategy` functions in the CoreRootRouter contract initiate cross-chain requests to the branchChain for updating the state of the BranchPort contract. In essence, invoking any of these functions in the CoreRootRouter contract on the rootChain triggers functions with identical names in the BranchPort of the branchChain as a result of cross-chain–message transferring.

However, an issue can arise because these corresponding functions in BranchPort include additional verification checks that are absent in the CoreRootRouter contract. Although the problem is not critical because these functions can only be invoked by the contract's owner, successfully sent cross-chain messages may be reverted, leading to unnecessary gas consumption and commission fees. To address this, we recommend implementing equivalent verification checks as found in the `_setPortStrategySettings` and `_setStrategyTokenMinimumReservesRatio` functions of the BranchPort contract.

4.2. Enhanced validation in `retryDepositSigned` and `retryDeposit` functions

In the current implementation of the `retryDepositSigned` function, the conditional check is implemented: `if (deposit.isSigned == UNSIGNED_DEPOSIT) revert`. This condition triggers a revert if the `deposit.isSigned` matches `UNSIGNED_DEPOSIT`. Presently, the contract recognizes only two distinct states: `SIGNED_DEPOSIT` and `UNSIGNED_DEPOSIT`. However, given that `isSigned` is defined as a `uint88`, there exists a theoretical possibility for it to acquire a value beyond these two predefined states.

To ensure robustness and accommodate any future expansions or unforeseen scenarios, it is advisable to modify the validation logic. A more comprehensive approach would be to implement the check as `if (deposit.isSigned != SIGNED_DEPOSIT) revert`. This modification ensures that the function only proceeds when the deposit is explicitly in the `SIGNED_DEPOSIT` state, thereby enhancing the system's resilience against potential anomalies or changes in state definitions.

This recommendation for enhanced validation logic is equally applicable to the `retryDeposit` function, where a similar pattern of state verification is observed. Implementing this change will contribute to the overall reliability and maintainability of the contract's codebase.

```
function retryDepositSigned(
       uint32 _depositNonce,
       bytes calldata _params,
       GasParams calldata _gParams,
       bool _hasFallbackToggled
    ) external payable override lock {
       // Get Settlement Reference
       Deposit storage deposit = getDeposit[_depositNonce];

       // Check if deposit is signed
       if (deposit.isSigned == UNSIGNED_DEPOSIT) revert NotDepositOwner();
       ...
    }
```

4.3. Inaccurate status update

The `lzReceiveNonBlocking` function in the BranchBridgeAgent contract in one of the cases processes the `Retrieve Settlement` payload data received from the Root chain. If the `executionState` is `STATUS_DONE`, the function reverts, indicating that the settlement has already been successfully executed. In contrast, if the `executionState[nonce]` is `STATUS_READY`, it updates the state to `STATUS_RETRIEVE`. The function then performs a fallback call to the Root chain.

However, the `executionState` mapping stores `uint256` values, theoretically allowing for states beyond the predefined `STATUS_READY`, `STATUS_DONE`, and `STATUS_RETRIEVE`. Therefore, if the current status is not `STATUS_READY`, the `executionState[nonce]` will remain unchanged. This means that the function does not guarantee the state to be `STATUS_RETRIEVE` in such cases, leaving room for additional states within the `executionState` mapping. Therefore, we recommend checking that the status is set to expected `STATUS_RETRIEVE` before performing the fallback call.

```
mapping(uint256 settlementNonce => uint256 state) public executionState;

function lzReceiveNonBlocking(
    address _endpoint,
    uint16 _srcChainId,
    bytes calldata _srcAddress,
    bytes calldata _payload
) public payable override requiresEndpoint(_srcChainId, _endpoint,
    _srcAddress) {
    //Save Action Flag
    bytes1 flag = _payload[0] & 0x7F;

    // Save settlement nonce
    uint32 nonce;
    ...
    //DEPOSIT FLAG: 4 (Retrieve Settlement)
        } else if (flag == 0x04) {
            // Parse recipient
            address payable recipient =
    payable(address(uint160(bytes20(_payload[PARAMS_START:PARAMS_START_SIGNED]))));
            //Get nonce
            nonce =
    uint32(bytes4(_payload[PARAMS_START_SIGNED:PARAMS_TKN_START_SIGNED]));
            //Check if settlement is in retrieve mode
            if (executionState[nonce] == STATUS_DONE) {
                revert AlreadyExecutedTransaction();
            } else {
                //Set settlement to retrieve mode, if not already set.
                if (executionState[nonce] == STATUS_READY)
    executionState[nonce] = STATUS_RETRIEVE;
                //Trigger fallback/Retry failed fallback
                _performFallbackCall(recipient, nonce);
            }
}
```

4.4. Test suite

While the overall test coverage of the project is good, there are some critical functionalities that have not been fully tested. Specifically, there is a lack of negative test cases, which are essential for ensuring the platform's resilience to unexpected inputs and edge cases. Moreover, some specific functions within the smart contract remain untested. For example:

- **BranchBridgeAgent.sol:** `callOutSigned`, `redeemDeposit`
- **RootBridgeAgent.sol:** `forceResumeReceive`, `redeemDeposit`
- **RootPort.sol:** `fetchVirtualAccount`
- **VirtualAccount.sol:** `call`, `payableCall`

As such, it is recommended that the development team focus on writing test cases for functionalities that have not been fully tested. These test cases can be relatively short and quick to write and execute, but they are essential for identifying potential vulnerabilities and ensuring that the platform is able to handle unexpected situations.

Because correctness is so critically important when developing smart contracts, we recommend that all projects strive for 100% code coverage. Testing should be an essential part of the software development lifecycle. No matter how simple a function may be, untested code is always prone to bugs. Furthermore, it is important to ensure that external contracts are also held to similar level of testing rigor.

The significance of comprehensive testing becomes even more critical given the project's nature as an omnichain liquidity protocol. This multi-chain context amplifies the potential impact of even minor unaddressed issues. Therefore, thorough testing is not just a best practice but an essential safeguard to ensure the robustness and reliability of the platform across various chains.

Several tests have been added to enhance the test suite between commits `53ade662` ↗ and `a703952e` ↗.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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.

Please note that our threat model was based on commit `35bee3b6` ↗, which represents a specific snapshot of the codebase. Therefore, it's important to understand that the absence of certain tests in our report may not reflect the current state of the test suite.

During the remediation phase, Maia DAO took proactive steps to address the findings by adding test cases where applicable. They also ensured that previously uncovered code branches were thoroughly tested with multiple tests. This demonstrates their dedication to enhancing the code quality and overall reliability of the system, which is commendable.

5.1. Module: BranchBridgeAgent.sol

Function: `callOutSignedAndBridge(bytes _params, DepositInput _dParams, GasParams _gParams, bool _hasFallbackToggled)`

The function allows the execution of a cross-chain request, which involves locking funds into the localPort contract. On the Root chain, the Virtual Account contract associated with the caller will receive the `_amount` of global `_hToken`. Additionally, the caller can provide extra payload data to trigger functions of the Virtual Account on the Root chain.

Inputs

- `_params`
  - **Control**: Full control by the caller.
  - **Constraints**: N/A.
  - **Impact**: Contains additional data for `rootRouterAddress.executeSignedDepositSingle` — can be empty.
- `_dParams`
  - **Control**: Full control by the caller, who must own the corresponding amount of tokens.
  - **Constraints**: The `localPort` contract must be able to burn `_dParams.hToken` tokens and must have approval from the caller to transfer `_dParams.token`.
  - **Impact**: It includes `_dParams.hToken`, `_dParams.token`, `_dParams.amount`, and `_dParams.deposit` — `_dParams.hToken` will be burned by the localPort contract, and `_dParams.token` will be transferred from the caller to the lo-

calPort contract.

- `_gParams`
    - **Control**: Full control by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: These values are used to encode `AdapterParameters` data for the LayerZero relayer contract.
- `_hasFallbackToggled`
    - **Control**: Full control by the caller.
    - **Constraints**: N/A.
    - **Impact**: If true, a fallback call will be performed after receiving the message in the Root chain.

Branches and code coverage

Intended branches

- Check that the deposit was successful.
    - ☑ Test coverage
- Check that the fallback call was performed.
    - ☑ Test coverage

Function call analysis

- `this._createDeposit(True, _depositNonce, msg.sender, _dParams.hToken, _dParams.token, _dParams.amount, _dParams.deposit) -> IBranchPort(this.localPortAddress).bridgeOut(msg.sender, _hToken, _token, _amount, _deposit)`
    - **What is controllable?** The `_dParams` is controlled by the caller.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function burns the `_dParams._amount - _dParams._deposit` amount of `_dParams.hToken` tokens from the caller's account. If `_dParams.token` is not the zero address, it transfers the `_deposit` amount of `_dParams.token` tokens from the caller to the localPortAddress contract address.
- `this._performCall(address payable(msg.sender), payload, _gParams, _hasFallbackToggled ? BridgeAgentConstants.BRANCH_BASE_CALL_OUT_SIGNED_DEPOSIT_SINGLE_GAS + BridgeAgentConstants.BASE_FALLBACK_GAS : BridgeAgentConstants.BRANCH_BASE_CALL_OUT_SIGNED_DEPOSIT_SINGLE_GAS) -> ILayerZeroEndpoint(this.lzEndpointAddress).send{value: msg.value}`
    - **What is controllable?** The payload is partly controlled; the caller controls `_dParams.hToken`, `_dParams.token`, `_dParams.amount`, and `_dParams.deposit`.

    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function can revert if the caller does not provide enough fee payment or if `rootChainId` is unknown. However, the caller does not control `rootChainId`.
- `this._performCall(address payable(msg.sender), payload, _gParams, _hasFallbackToggled ? BridgeAgentConstants.BRANCH_BASE_CALL_OUT_SIGNED_DEPOSIT_SINGLE_GAS + BridgeAgentConstants.BASE_FALLBACK_GAS : BridgeAgentConstants.BRANCH_BASE_CALL_OUT_SIGNED_DEPOSIT_SINGLE_GAS) -> IRootBridgeAgent(this.rootBridgeAgentAddress).lzReceive{value: msg.value}`
    - **What is controllable?** The payload is partly controlled by the initial caller. The initial caller controls `_dParams.hToken`, `_dParams.token`, `_dParams.amount`, and `_dParams.deposit`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** This function call is a part of the cross-chain–message transferring process, so it will be executed in a different transaction and on a different chain than the original call. Since all the logic for message processing is executed within the `excessivelySafeCall` function, in case of a revert, the message will not be saved for resending and all native tokens from this contract will be transferred to the rootPortAddress contract. But if either transfers revert, the message will be saved for resending. However, it will also be possible to skip this message using the `forceResumeReceive` function. During payload processing, the function can revert if `_dParams.hToken` is not set as a trusted local token in `_srcChainId` and if `_dParams.token` is not set as the underlying token for `_dParams.hToken` from `_srcChainId`.

Function: `callOutSigned(bytes _params, GasParams _gParams)`

This transfers a cross-chain message using the LayerZero protocol. This allows a user to invoke their VirtualAccount contract on the Root chain. Importantly, this message does not allow to make a deposit. The message will be processed by the `lzReceiveNonBlocking` function of the RootBridgeAgent contract upon delivery to the Root chain. Subsequently, the user's `_params` will be forwarded to the `executeSigned` function of the rootRouter contract. The processing of `_params` varies based on the implementation of the rootRouter contract. The MulticallRootRouter contract enables the user to execute one of three actions:

- Invoke the `call` function of the VirtualAccount contract.
    - Invoke the `call` function, withdraw assets from the Virtual Account contract, and bridge out the withdrawn assets.
    - Invoke the `call` function, withdraw a batch of token assets, and bridge them out.

Inputs

- `_params`
    - **Control**: Full control by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: Contains the data that will be processed in the RootRouter contract on the Root chain. In the case of `MulticallRootRouter`, the first byte should be the ID of the function (0x01, 0x02, 0x03; in other cases, `_executeSigned` will revert). The rest of the data depends on the type of function.
- `_gParams`
    - **Control**: Full control by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: These values are used to encode `AdapterParameters` data for the LayerZero relayer contract.

Branches and code coverage

No tests have been implemented for this function.

Intended branches

- Check if the branch has test coverage.
    - ☐ Test coverage

Negative behavior

- The incorrect `_gParams` data.
    - ☐ Negative test

Function call analysis

- `this._performCall(address payable(msg.sender), payload, _gParams, BridgeAgentConstants.BRANCH_BASE_CALL_OUT_SIGNED_GAS) -> ILayerZeroEndpoint(lzEndpointAddress).send{value: msg.value}`
    - **What is controllable?** The `_payload` is partially controlled, `_params` is fully controlled by the caller, and `msg.value` is used as a fee for cross-chain transferring.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The `callOutSigned` function has a `lock` to prevent reentrancy attacks. The function can revert if the caller does not provide sufficient fee payment.
- `this._performCall(address payable(msg.sender), payload, _gParams,`

`BridgeAgentConstants.BRANCH_BASE_CALL_OUT_SIGNED_GAS) -> IRootBridgeAgent(this.rootBridgeAgentAddress).lzReceive{value: msg.value}`

- **What is controllable?** The `_payload` is partly controlled by the initial caller in srcChain.
- **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
- **What happens if it reverts, reenters or does other unusual control flow?** This function call is a part of the cross-chain–message transferring process, but it will be executed in a different transaction and on a different chain than the original call. Since all the logic for message processing is executed within the `excessivelySafeCall` function, in case of a revert, the message will not be saved for resending and all native tokens from this contract will be transferred to the rootPortAddress contract. But if either transfers revert, the message will be saved for resending. However, it will also be possible to skip this message using the `forceResumeReceive` function.

Function: `redeemDeposit(uint32 _depositNonce, address _recipient, address _localTokenAddress)`

The function allows for the partial redemption of deposits, specifically for `hTokens` that are equivalent to the `_localTokenAddress` provided by the caller. To facilitate this, several conditions must be met: the deposit corresponding to the `_depositNonce` must exist, the caller must be the owner of the deposit, and the `retrieveDeposit` function must be executed and successfully delivered to the Root chain, executed there, and responded back to this chain to change the deposit's status to `STATUS_FAILED`. After successfully transferring tokens to the `_recipient`, the deposit information for these tokens will be removed. However, the complete deposit data will only be deleted from storage after the full redemption of all tokens. Therefore, it will be possible to call this function again if `deposit.tokens.length` is not zero.

Inputs

- `_depositNonce`
  - **Control**: The caller controls the `_depositNonce`.
  - **Constraints**: The caller must be the owner of the deposit object related to this `_depositNonce`.
  - **Impact**: Determines which deposit object is being accessed for redemption.
- `_recipient`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The address of the receiver of the deposited tokens.
- `_localTokenAddress`
  - **Control**: Full control.

- **Constraints**: The caller must have the same `hToken` in the deposit object as `_localTokenAddress` to perform redemption.
- **Impact**: The deposit will be partially redeemed only for the corresponding `_localTokenAddress`.

Branches and code coverage

No tests have been implemented for this function.

Function call analysis

- `this._clearToken(_recipient, deposit.hTokens[i], deposit.tokens[i], deposit.amounts[i], deposit.deposits[i]) -> IBranchPort(this.localPortAddress).bridgeIn(_recipient, _hToken, _amount - _deposit)`
  - **What is controllable?** The `_recipient` is under the caller's control, while the rest of the data is sourced from an existing deposit owned by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if the BranchPort fails to mint `_hToken` tokens.
- `this._clearToken(_recipient, deposit.hTokens[i], deposit.tokens[i], deposit.amounts[i], deposit.deposits[i]) -> IBranchPort(this.localPortAddress).withdraw(_recipient, _token, _deposit)`
  - **What is controllable?** The `_recipient` is under the caller's control, while the rest of the data is sourced from an existing deposit owned by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if the BranchPort does not have a sufficient amount of the underlying `_token`.

Function: `redeemDeposit(uint32 _depositNonce, address _recipient)`

The function allows for the redemption of a deposit. To facilitate this, several conditions must be met: the deposit corresponding to the `_depositNonce` must exist, the caller must be the owner of the deposit, and the `retrieveDeposit` function must be executed. This function must then be delivered to the Root chain, successfully executed there, and a response sent back to this chain to change the deposit's status to `STATUS_FAILED`. After successfully transferring tokens to the `_recipient`, the deposit object will be deleted to prevent the possibility of redeeming the same deposit twice.

Inputs

- `_depositNonce`
  - **Control**: The caller controls the `_depositNonce`.
  - **Constraints**: The caller must be the owner of the deposit object related to this `_depositNonce`.
  - **Impact**: Determines which deposit object is being accessed for redemption.
- `_recipient`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The address of the receiver of the deposited tokens.

Branches and code coverage

Intended branches

- Check that the deposit was deleted after redeeming.
  - ☑ Test coverage

Negative behavior

- Double redeem is not possible.
  - ☑ Negative test

Function call analysis

- `this._clearToken(_recipient, deposit.hTokens[i], deposit.tokens[i], deposit.amounts[i], deposit.deposits[i]) -> IBranchPort(this.localPortAddress).bridgeIn(_recipient, _hToken, _amount - _deposit)`
  - **What is controllable?** The `_recipient` is under the caller's control, while the rest of the data is sourced from an existing deposit owned by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if the BranchPort fails to mint `_hToken` tokens.
- `this._clearToken(_recipient, deposit.hTokens[i], deposit.tokens[i], deposit.amounts[i], deposit.deposits[i]) -> IBranchPort(this.localPortAddress).withdraw(_recipient, _token, _deposit)`
  - **What is controllable?** The `_recipient` is under the caller's control, while the rest of the data is sourced from an existing deposit owned by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?**

The function will revert if the BranchPort does not have a sufficient amount of the underlying `_token`.

Function: `retrieveDeposit(uint32 _depositNonce, GasParams _gParams)`

The function sends a cross-chain request to set the status of a deposit to `STATUS_FAILED` if the deposit was not successfully executed on the Root chain. To initiate this, several conditions must be met: the deposit associated with `_depositNonce` must exist, the caller must be the owner of this deposit, and the deposit's status should not already be `STATUS_FAILED`.

Inputs

- `_depositNonce`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The caller must be the owner of the deposit related to this `_depositNonce`.
  - **Impact**: The status of this deposit will be changed to `STATUS_FAILED` if the deposit action was not successfully executed on the Root chain. Subsequently, the owner of the deposit will be able to redeem these deposited funds.
- `_gParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: There are no constraints.
  - **Impact**: These values are used to encode `AdapterParameters` data for the LayerZero relayer contract.

Branches and code coverage

Intended branches

- Check that status was changed to `STATUS_FAILED`.
  - [ ] Test coverage

Negative behavior

- The deposit for `_depositNonce` does not exist.
  - [ ] Negative test

Function call analysis

- `this._performCall(address payable(msg.sender), payload, _gParams, BridgeAgentConstants.BRANCH_BASE_CALL_OUT_GAS) -> ILayerZeroEndpoint(this.lzEndpointAddress).send{value: msg.value}`

    - **What is controllable?** The caller controls `_depositNonce` and must be the owner of this deposit; additionally, the `_gParams` parameters are controlled by the caller.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function can revert if the provided fee is insufficient.
- `this._performCall(address payable(msg.sender), payload, _gParams, BridgeAgentConstants.BRANCH_BASE_CALL_OUT_GAS) -> IRootBridgeAgent(this.rootBridgeAgentAddress).lzReceive{value: msg.value}`
    - **What is controllable?** The `payload` contains the address of the initial caller and the `_depositNonce` provided and owned by the caller.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** If the `address(this).excessivelySafeCall` call is unsuccessful, it will return a `success` status of false, and all ETH balance of the contract will be transferred to the rootPortAddress contract. Therefore, the `lzReceive` function can only revert if the `safeTransferAllETH` function fails during the transfer of funds to the `rootPortAddress`.

Function: `retrySettlement(uint32 _settlementNonce, bytes _params, GasParams[Literal(value=2, unit=None)] _gParams, bool _hasFallbackToggled)`

This allows for the retry of settlement execution. The virtual account associated with the caller must be the owner of the settlement. (This check is performed in the RootBridgeAgent contract during the processing of this cross-chain request.)

Inputs

- `_settlementNonce`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: Represents the nonce of the settlement on the Root chain.
- `_params`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: N/A.
- `_gParams`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: These values are used to encode `AdapterParameters` data for the

LayerZero relayer contract.

- `_hasFallbackToggled`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: If true, a fallback call will be executed after settlement execution in this chain.

Branches and code coverage

Intended branches

- Check that status of settlement was changed in BranchBridgeAgent.
    - ☐ Test coverage
- Check that status of settlement was changed in RootBridgeAgent if the fallback was executed.
    - ☐ Test coverage

Negative behavior

- The settlement already executed.
    - ☐ Negative test
- The caller is not an owner of settlement.
    - ☐ Negative test

Function call analysis

- `this._performCall(address payable(msg.sender), payload, _gParams[0], BridgeAgentConstants.BRANCH_BASE_CALL_OUT_GAS) -> ILayerZeroEndpoint(this.lzEndpointAddress).send{value: msg.value}`
    - **What is controllable?** The `_payload` is fully controlled by the caller, except for the address of the caller and the execution flag.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function can revert if the provided fee is insufficient.
- `this._performCall(address payable(msg.sender), payload, _gParams[0], BridgeAgentConstants.BRANCH_BASE_CALL_OUT_GAS) -> IRootBridgeAgent(this.rootBridgeAgentAddress).lzReceive{value: msg.value}`
    - **What is controllable?** The `_payload` is fully controlled by the initial caller, except for the address of the caller (since `msg.sender` is used, it cannot be an arbitrary address) and the execution flag.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are checks to ensure that the initial caller is the settlement owner

and that the settlement has not already been redeemed (the settlement object for the provided nonce still exists). After these checks, cross-chain `executeWithSettlement` requests will be sent to the Branch chain.

5.2. Module: BranchPort.sol

Function: `manage(address _token, uint256 _amount)`

The function allows a strategy contract to withdraw a specified amount of a given token. The function performs several checks: ensure that strategy is trusted contract and that the requested amount does not exceed the limit.

Inputs

- `_token`
    - **Control**: Full control by the caller.
    - **Constraints**: The `_token` should be a trusted strategy token.
    - **Impact**: Specifies the token that will be transferred from this contract to the caller account.
- `_amount`
    - **Control**: Full control by the caller.
    - **Constraints**: There is a check to ensure that the requested amount does not exceed limits.
    - **Impact**: The amount of the token requested by strategy.

Branches and code coverage

Intended branches

- Check that tokens have been transferred to the strategy.
    - ☑ Test coverage
- Check that `getPortStrategyTokenDebt` has been increased.
    - ☑ Test coverage

Negative behavior

- Caller is untrusted strategy.
    - ☐ Negative test
- The `_amount` is more than the limit.
    - ☐ Negative test

Function call analysis

- `SafeTransferLib.safeTransfer(_token, msg.sender, _amount)`
  - **What is controllable?** `_token` and `_amount` are controlled by the caller,
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** Function transfers the requested tokens to the strategy contract. If the contract does not have enough tokens, the function call will revert.

Function: `replenishReserves(address _token, uint256 _amount)`

This function allows a strategy contract to repay a specified amount of debt with a given token.

Inputs

- `_token`
  - **Control**: Full control by the caller.
  - **Constraints**: There are no constraints.
  - **Impact**: Specifies the token in which the debt is being repaid.
- `_amount`
  - **Control**: Full control by the caller.
  - **Constraints**: The function will revert if the strategy does not have enough debt to repay this amount.
  - **Impact**: Represents the amount of the token to be repaid.

Branches and code coverage

Intended branches

- Check that tokens have been transferred from the caller.
  - ☑ Test coverage
- Check that `getPortStrategyTokenDebt` has been decreased.
  - ☐ Test coverage

Negative behavior

- Caller is not trusted strategy.
  - ☐ Negative test
- The `_amount` is more than the debt for this strategy.
  - ☐ Negative test

Function call analysis

- `IPortStrategy(msg.sender).withdraw(address(this), _token, _amount)`
    - **What is controllable?** `_token` and `_amount` are controlled by the caller, but if `_amount` is not zero, the function will revert if the caller/strategy does not have enough debt to repay this amount.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `replenishReserves(address _strategy, address _token)`

This function allows any caller to repay the reserves lacking of `_token` tokens from the `_strategy` contract.

Inputs

- `_strategy`
    - **Control**: Full control by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: The contract of the strategy from which funds will be withdrawn to fill the reserves lacking but no more than `reservesLacking`.
- `_token`
    - **Control**: Full control by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: Specifies the token in which the debt is being repaid.

Branches and code coverage

Intended branches

- Check that tokens have been transferred from the `_strategy`.
    - ☑ Test coverage
- Check that `getPortStrategyTokenDebt` has been decreased.
    - ☐ Test coverage

Negative behavior

- The `_strategy` is not trusted.
    - ☐ Negative test

Function call analysis

- `IPortStrategy(_strategy).withdraw(address(this), _token, amountToWithdraw)`
    - **What is controllable?** The `_token` is controlled by the caller.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function has a lock against reentrancy attacks. The `_strategy` contract is not guaranteed to be trusted.

5.3. Module: RootBridgeAgent.sol

Function: `forceResumeReceive(uint16 _srcChainId, bytes _srcAddress)`

This allows anyone to call this function to delete stored payload to force resume cross-chain messaging communication.

Inputs

- `_srcChainId`
    - **Control**: Full control by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: The `storedPayload` associated with this `_srcChainId` and `_srcAddress` will be deleted from the lzEndpoint contract.
- `_srcAddress`
    - **Control**: Full control by the caller.
    - **Constraints**: There are no constraints.
    - **Impact**: The `storedPayload` associated with this `_srcChainId` and `_srcAddress` will be deleted from the lzEndpoint contract.

Branches and code coverage

No tests have been implemented for this function.

Function call analysis

- `ILayerZeroEndpoint(this.lzEndpointAddress).forceResumeReceive(_srcChainId, _srcAddress)`
    - **What is controllable?** `_srcChainId` and `_srcAddress` are controlled by the caller.

- **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
- **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if there is not stored payload for `_srcAddress` and `_srcChainId` in the lzEndpoint contract.

Function: `redeemSettlement(uint32 _settlementNonce, address _recipient)`

This function allows the caller to redeem the assets associated with the settlement. The function ensures that the caller is authorized to redeem this settlement. It transfers each global token associated with this settlement to the specified recipient. After the redemption process is complete, the settlement is deleted from storage.

Inputs

- `_settlementNonce`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The function requires that the settlement corresponding to this nonce exists and is in a state that allows redemption.
  - **Impact**: Specifies the particular settlement to be redeemed.
- `_recipient`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: There are no constraints.
  - **Impact**: The function will transfer the redeemed assets to this address.

Branches and code coverage

Intended branches

- Check that the settlement was deleted.
  - ☑ Test coverage
- Check that the assets have been successfully transferred to the receiver if `_dstChainId` != `localChainId`.
  - ☐ Test coverage
- Check that the assets have been successfully transferred to the receiver if `_dstChainId` == `localChainId`.
  - ☐ Test coverage

Negative behavior

- The status of the settlement is not `STATUS_FAILED`.
  - ☐ Negative test
- The caller is not the owner of the settlement.

- ☐ Negative test

Function call analysis

- `IRootPort(this.rootPortAddress).bridgeToRoot(_recipient, IRootPort(this.rootPortAddress).getGlobalTokenFromLocal(_hToken, _dstChainId), settlement.amounts[i], settlement.deposits[i], _dstChainId)`
  - **What is controllable?** The caller controls only the `_recipient` address.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if `deposit` is more than `amount` — also if the RootPort contract fails to mint tokens and if the RootPort contract has an insufficient amount of tokens to transfer to the `_recipient` account.
- `IRootPort(this.rootPortAddress).getGlobalTokenFromLocal(_hToken, _dstChainId)`
  - **What is controllable?** The caller does not control the parameters directly.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns the address of the token in this chain associated with `_hToken` in srcChain. The caller does not control this association.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `getGlobalToken` was not set for this `_hToken`, the function will return the zero address.

Function: `retrieveSettlement(uint32 _settlementNonce, GasParams _gParams)`

The `retrieveSettlement` function allows to retrieve a specific settlement identified by `_settlementNonce`. This function is callable by any user. The function ensures that the caller is authorized to retrieve the settlement by verifying the settlement owner's address. After these validations, it performs a cross-chain call to update the status of the settlement to `STATUS_FAILED` if this settlement is not executed in the `dstChainId`.

Inputs

- `_settlementNonce`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The function checks that the settlement corresponding to this `_settlementNonce` exists and has not already been retrieved.
  - **Impact**: Specifies the particular settlement to be retrieved.
- `_gParams`
  - **Control**: Fully controlled by the caller.

- **Constraints**: There are no constraints.
- **Impact**: These values are used to encode `AdapterParameters` data for the LayerZero relayer contract.

Branches and code coverage

Intended branches

- The settlement status was updated to `STATUS_FAILED` after the fallback call.
  - ☑ Test coverage

Negative behavior

- Caller is not the owner of the settlement.
  - ☐ Negative test
- The settlement does not exist.
  - ☐ Negative test

Function call analysis

- `this._checkSettlementOwner(msg.sender, settlement.owner) -> IRootPort(this.rootPortAddress).getUserAccount(settlementOwner)`
  - **What is controllable?** Nothing is controlled by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns the address of the user account. If the user account has not been deployed yet, the function returns the zero address.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._performCall(settlement.dstChainId, address payable(settlementOwner), payload, _gParams, BridgeAgentConstants.ROOT_BASE_CALL_OUT_GAS) -> ILayerZeroEndpoint(this.lzEndpointAddress).send{value: msg.value}`
  - **What is controllable?** The `_settlementNonce` from the payload is controlled by the caller, but the caller must be the owner of this settlement.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function can revert if the caller does not provide enough fee payment or if the `dstChainId` is unknown.
- `this._performCall(settlement.dstChainId, address payable(settlementOwner), payload, _gParams, BridgeAgentConstants.ROOT_BASE_CALL_OUT_GAS) -> IBranchBridgeAgent(callee).lzReceive{value: msg.value}`
  - **What is controllable?** The `_settlementNonce` from the payload is controlled by the initial caller, but the initial caller must be the owner of this

settlement.

- **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
- **What happens if it reverts, reenters or does other unusual control flow?** If the `executionState` for the `_settlementNonce` is equal to `STATUS_DONE`, the function will revert. If the `executionState` is equal to `STATUS_READY`, the `executionState` will be changed to the `STATUS_RETRIEVE`; otherwise, it will remain the same. Subsequently, a `FallbackCall` will be performed back to the Root chain to change the `getSettlement[nonce].status` to `STATUS_FAILED`.

5.4. Module: RootPort.sol

Function: `fetchVirtualAccount(address _user)`

The `fetchVirtualAccount` function allows any caller to associate a Virtual Account contract with a specified `_user` address. If a Virtual Account for the given `_user` does not already exist, the function deploys a new Virtual Account contract and updates the `getUserAccount` mapping with this new contract address. If a Virtual Account already exists for the `_user`, the function simply returns the existing account without making any changes.

Inputs

- `_user`
    - **Control**: Full control by the caller.
    - **Constraints**: If a Virtual Account contract is already deployed for `_user`, the function will not deploy a new one and will return the existing account.
    - **Impact**: Determines the user address for which the Virtual Account is fetched or deployed.

Branches and code coverage

No separate tests have been implemented for this function.

5.5. Module: VirtualAccount.sol

Function: `call(Call[] calls)`

This aggregates calls, ensuring each call is successful. Inspired by the Multicall2 contract.

Inputs

- `calls`
    - **Control**: Full control. Consists of (address, calldata) pairs.
    - **Constraints**: Addresses must belong to contracts. Failing calls will revert.
    - **Impact**: The call(s) to make.

Branches and code coverage

Intended branches

- Approved caller makes a single call.
    - ☐ Test coverage
- Approved caller makes an aggregated call.
    - ☐ Test coverage

Negative behavior

- Unapproved caller tries to make a call.
    - ☑ Negative test
- Tries to call a non-contract.
    - ☐ Negative test
- One of multiple calls fail.
    - ☐ Negative test

Function: `payableCall(PayableCall[] calls)`

This aggregates calls with a message value, ensuring each call is successful. Inspired by the Multicall3 contract.

Inputs

- `calls`
    - **Control**: Full control. Consists of (target, calldata, value) triples.
    - **Constraints**: Target must be a contract. Total value must equal `msg.value` exactly.
    - **Impact**: The targets to send a payable call to and the data to include.

Branches and code coverage

Intended branches

- Approved caller makes a single call with a payment.
    - ☑ Test coverage

- Approved caller makes an aggregated call with payments.
  - ☐ Test coverage

Negative behavior

- Unapproved caller tries to make a call.
  - ☑ Negative test
- Tries to call a non-contract.
  - ☑ Negative test
- One of multiple calls fail.
  - ☐ Negative test
- Accumulated value differs from `msg.value`.
  - ☑ Negative test

Function: `withdrawERC20(address _token, uint256 _amount)`

This withdraws ERC-20–based tokens from the virtual account to the sender. Can only be called by the approved caller.

Inputs

- `_token`
  - **Control**: Full control.
  - **Constraints**: None, but it reverts later if the address is not a valid ERC-20 token.
  - **Impact**: The address of the ERC-20 token to withdraw.
- `_amount`
  - **Control**: Full control.
  - **Constraints**: None, but it reverts later if more than the current token balance.
  - **Impact**: The amount of ERC-20 tokens to withdraw.

Branches and code coverage

Intended branches

- Approved caller can withdraw.
  - ☑ Test coverage

Negative behavior

- Unapproved caller cannot withdraw.
  - ☑ Negative test

Function: `withdrawERC721(address _token, uint256 _tokenId)`

Transfers ownership of an ERC-721 token (NFT) from the virtual account to the sender. Can only be called by the approved caller.

Inputs

- `_token`
    - **Control**: Full control.
    - **Constraints**: None, but it will revert if specifying a contract that does not implement ERC-721.
    - **Impact**: The address of the token contract to withdraw from.
- `_tokenId`
    - **Control**: Full control.
    - **Constraints**: None, but it will revert if the token ID is not owned by the virtual account nor approved to transfer it.
    - **Impact**: The token ID to withdraw.

Branches and code coverage

Intended branches

- Transfer NFT to the approved caller.
    - ☐ Test coverage

Negative behavior

- Unapproved caller tries to claim NFT.
    - ☑ Negative test

Function: `withdrawNative(uint256 _amount)`

This withdraws native ETH from the virtual account to the sender. Can only be called by the approved caller.

Inputs

- `_amount`
    - **Control**: Full control.
    - **Constraints**: None, but it reverts later if more than the current ETH balance.
    - **Impact**: The amount of ETH to withdraw.

Branches and code coverage

Intended branches

- Approved caller can withdraw.
  - ☑ Test coverage

Negative behavior

- Unapproved caller cannot withdraw.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Maia DAO Ulysses Protocol contracts, we discovered two findings, both of which were low impact. 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.