Zellic

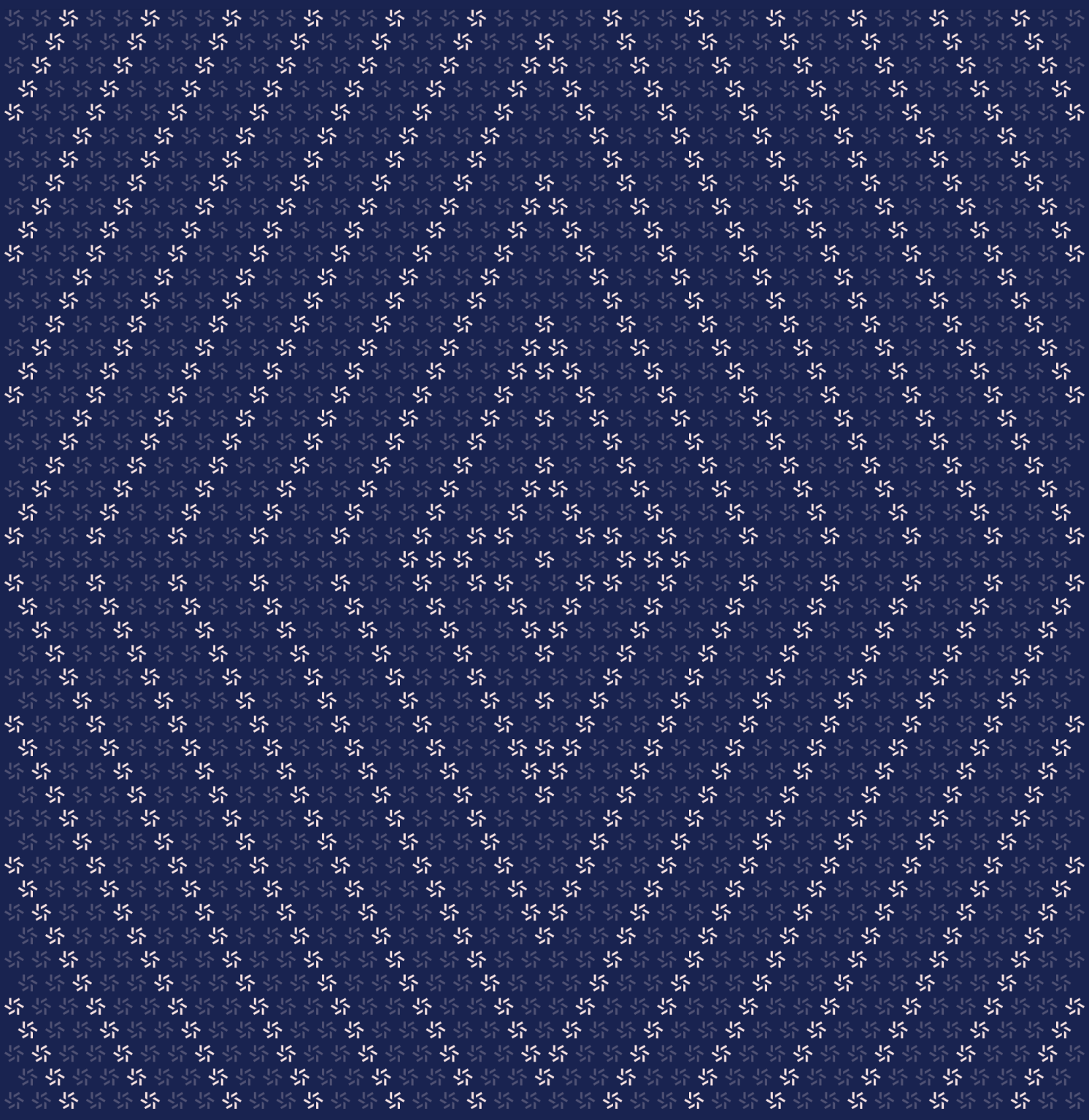
Prepared for
Joshua Baker
Wyatt Cravatas
Aori

Prepared by
Katerina Belotskaia
Qingying Jie
Zellic

May 1, 2025

Aori

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 information security 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. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Aori from April 21st to April 24th, 2025. During this engagement, Zellic reviewed Aori's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- Are there potential race conditions or state inconsistencies in cross-chain settlement flows?
- Could the hook mechanism be exploited through reentrancy or other attack vectors?
- Are there edge cases in the balance tracking that could lead to permanently locked funds?
- How robust is the cross-chain message validation against forgery or replay attacks?

1.3. 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.

1.4. Results

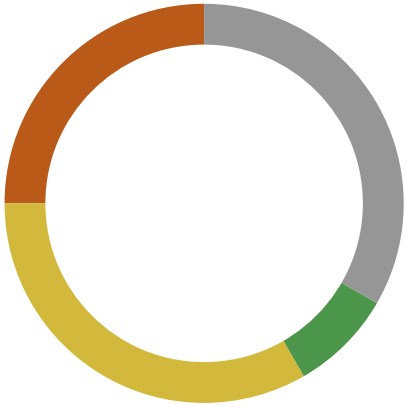
During our assessment on the scoped Aori contracts, we discovered 12 findings. No critical issues were found. Three findings were of high impact, four were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Aori in the Discussion section (4. ↗).

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	4
Low	1
Informational	4



2. Introduction

2.1. About Aori

Aori contributed the following description of Aori:

> Aori is an intent settlement protocol that enables secure token exchanges across different blockchains between users and trusted solvers. The system allows users to deposit tokens on a source chain with signed intent parameters, which solvers (market makers) can fulfill on destination chains. The protocol manages the full intent lifecycle through secure token custody, EIP-712 signature verification, and LayerZero messaging for cross-chain settlement, ensuring intents are executed according to signed parameters.

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:

Aori Contracts

Type	Solidity
Platform	EVM-compatible
Target	aori
Repository	https://github.com/aori-io/aori ↗
Version	`09826f813ce032f40146399e8ee58eb6134bdc86`
Programs	`Aori.sol` `AoriUtils.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.2 person-weeks. The assessment was conducted by two consultants over the course of four calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io

Qingying Jie
Engineer
qingying@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 21, 2025	Start of primary review period
April 22, 2025	Kick-off call
April 24, 2025	End of primary review period

3. Detailed Findings

3.1. The `settle` message can be undeliverable due to incorrect `extraOptions` configuration

Target	Aori		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `settle` function allows any caller to initiate the process of transferring a cross-chain message that contains a batch of filled orders. This cross-chain transfer is performed using the LayerZero (LZ) protocol. The caller provides the prepared `payload` and `extraOptions`, which are passed to the `endpoint.send()` function to trigger the transfer.

The `extraOptions` parameter gives the caller control over message configuration, including the gas amount and `msg.value` that the executor will use during message delivery.

```
function settle(
    uint32 srcEid,
    address filler,
    bytes calldata extraOptions
) external payable nonReentrant whenNotPaused {
    bytes32[] storage arr = srcEidToFillerFills[srcEid][filler];
    [...]
    _lzSend(srcEid, payload, extraOptions, MessagingFee(msg.value, 0),
    payable(msg.sender));
    emit SettleSent(srcEid, filler, payload);
}
function _lzSend(
    uint32 _dstEid,
    bytes memory _message,
    bytes memory _options,
    MessagingFee memory _fee,
    address _refundAddress
) internal virtual returns (MessagingReceipt memory receipt) {
    [...]
    return
        // solhint-disable-next-line check-send-result
        endpoint.send{ value: messageValue }(
            MessagingParams(_dstEid, _getPeerOrRevert(_dstEid), _message,
    _options, _fee.lzTokenFee > 0),
            _refundAddress
```

```
        );
}
```

Impact

Because there are no restrictions on who can call the `settle` function and no enforced validation of `extraOptions`, a malicious caller could specify insufficient gas or a manipulated configuration in `extraOptions`.

This may result in settlement messages being undeliverable on the destination chain if the executor is unable to complete message delivery due to improper gas configuration.

Recommendations

We recommend the following.

1. Restrict access to the `settle` function by applying an `onlySolver` modifier, ensuring only trusted solvers can initiate cross-chain order settlement.

2. Use enforced options by integrating LayerZero's OAppOptionsType3 ↗ contract. This will allow to define `enforcedOptions` for each message type and endpoint, ensuring consistent and safe configuration for all messages, regardless of caller-provided `extraOptions`.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit a7ee7af8 ↗.

Aori provided the following response to this finding:

> Added onlySolver to settle() function. While configuring the layerZero options on chain using OAppOptionsType3 is on the roadmap, our offchain infrastructure is configured properly to call quote() before submitting cancel() or settle() calls that trigger LayerZero Messages with proper extraOptions configuration. We plan to finish on chain LZ options configuration before opening up the protocol to additional solvers to ensure no LZ messages are made undeliverabe due to misconfigured solver infrastructure.

3.2. Double charges in the single-chain swaps

Target	Aori		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

During a deposit with a hook to create a single-chain swap, the offerer is charged input tokens twice. One is transferred to the hook address in the function `_executeSrcHook`; the other is deducted from the offerer's locked balance in the function `_settleSingleChainSwap`. Meanwhile, the function `_settleSingleChainSwap` uses the function `validateBalanceTransferOrRevert` to ensure that the offerer's locked input token balance is successfully deducted.

```
function deposit(
    Order calldata order,
    bytes calldata signature,
    SrcHook calldata hook
) external payable nonReentrant whenNotPaused onlySolver {
    // [...]
    (uint256 amountReceived, address tokenReceived) =
        _executeSrcHook(order, hook);

    if (order.isSingleChainSwap()) {
        // Save the order details
        orders[orderId] = order;

        // Settle the order immediately for single-chain swaps
        _settleSingleChainSwap(orderId, order, msg.sender, amountReceived);
    }
    // [...]
}

function _executeSrcHook(
    Order calldata order,
    SrcHook calldata hook
) internal allowedHookAddress(hook.hookAddress) returns (
    // [...]
) {
    // Transfer input tokens to the hook
    IERC20(order.inputToken).safeTransferFrom(
        order.offerer,
```

```
            hook.hookAddress,
            order.inputAmount
        );
        // [...]
    }

    function _settleSingleChainSwap(
        // [...]
    ) internal {
        // [...]
        if (balances[order.offerer][order.inputToken].locked >= order.inputAmount)
        {
            // Unlock the tokens from offerer's balance
            balances[order.offerer][order.inputToken].decreaseLockedNoRevert(
                uint128(order.inputAmount)
            );
            // [...]
        }
        // [...]
        balances[order.offerer][order.inputToken].validateBalanceTransferOrRevert(
            initialOffererLocked,
            finalOffererLocked,
            initialSolverUnlocked,
            finalSolverUnlocked,
            uint128(order.inputAmount)
        );
    }
```

Impact

The offerer may need to pay more tokens to create a single-chain order. Additionally, to lock tokens, the offerer must deposit them — that is, an order will be created in the contract. In this issue, charging those locked tokens would affect the other orders.

Recommendations

In the function `_settleSingleChainSwap`, consider transferring the locked input token according to the situation.

Remediation

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

- b7666afb ↗

- e0dceff8 ↗

Aori provided the following response to this finding:

> Since the requirement that the hook returns at least the outputAmount is enforced in _executeSrcHook, the cleanest approach here was to handle the transfers in the isSingleChainSwap path of _executeSrcHook, and reserve the _settleSingleChainSwap function to handle balance observations and internal accounting for just the withoutHook path.

3.3. Potential contract drain when filling a single-chain swap

Target	Aori		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

When filling a single-chain swap, the recipient can receive twice the `order.outputAmount` of `order.outputToken`. One is transferred from the solver; the other is from the contract.

```
function fill(Order calldata order)
    external payable nonReentrant whenNotPaused onlySolver {
    // [...]
    IERC20(order.outputToken).safeTransferFrom(msg.sender, order.recipient,
        order.outputAmount);

    // single-chain swap path
    if (order.isSingleChainSwap()) {
        uint256 amountReceived = order.outputAmount;
        _settleSingleChainSwap(orderId, order, msg.sender, amountReceived);
        return;
    }
    // [...]
}

function _settleSingleChainSwap(
    // [...]
) internal {
    // [...]

    // Transfer the output token to the recipient

    IERC20(order.outputToken).safeTransfer(order.recipient, order.outputAmount)
        ;

    // [...]
}
```

Impact

Since one of the `order.outputAmount` is transferred from the contract if the contract has sufficient `order.outputToken`, this can be used to drain the specified token from the contract.

Recommendations

In the function `_settleSingleChainSwap`, consider transferring the output token according to the situation.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `e0dceff8` ↗.

3.4. Race condition between `settle` and `cancel` can lead to filler loss

Target	Aori		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

During the order-settlement process, the `order` is first filled on the destination chain, and then a settlement payload, containing the `filler` address and `orderId`, is sent to the source chain via the `settle` function using the LayerZero protocol.

On the source chain, this payload is received and handled by the `_settleOrder` function. However, if the referenced `orderId` is no longer in an `Active` state, the settlement is silently skipped. This creates a situation where repeat receiving is impossible, because of two reasons:

1. On the destination chain, the order has already been marked as `Filled` and cannot be refilled or resettled.
2. On the source chain, the settlement payload cannot be retried.

```
function _settleOrder(bytes32 orderId, address filler) internal {
    if (orderStatus[orderId] != IAori.OrderStatus.Active) {
        return; // Any reverts are skipped
    }
    [...]
```

Additionally, the source chain provides a `cancel` function that allows a trusted solver to cancel any `Active` order, making the `offerer`'s input tokens withdrawable. If the order is canceled before the settlement message is delivered, the incoming payload will no longer be applicable and will be skipped.

```
function cancel(bytes32 orderId) external whenNotPaused {
    require(
        isAllowedSolver[msg.sender],
        "Only whitelisted solver can cancel from the source chain"
    );
    _cancel(orderId);
}
function _cancel(bytes32 orderId) internal {
    require(orderStatus[orderId] == IAori.OrderStatus.Active, "Can only cancel
```

```
        active orders");
        orderStatus[orderId] = IAori.OrderStatus.Cancelled;

        Order memory order = orders[orderId];

        balances[order.offerer][order.inputToken].unlock(uint128(order.inputAmount));

        emit CancelSent(orderId);
    }
```

Impact

If the settlement of the order is skipped due to the order no longer being in an `Active` state (e.g., because it was canceled), the filler may be unable to receive the corresponding output tokens from the source chain. Although the contract provides an `emergencyWithdraw` mechanism in both destination and source chains that allows the owner to recover stuck funds, this does not directly compensate the filler.

Recommendations

Rather than transferring output tokens immediately to the recipient during order fulfillment on the destination chain, consider crediting the recipient's internal balance in the contract. Funds should become withdrawable only after either successful settlement confirmation on the source chain or, if no settlement rejection is received within a reasonable timeout, the funds would then be released to the recipient.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit 9207ed20 ↗.

Aori provided the following response to this finding:

> Limited performance of emergency source chain cancellations to onlyOwner by adding a emergencyCancel() function. Mitigated potential race conditions by adding stricter validation around source chain cancellations with a validateSourceChainCancel function in AoriUtils.sol which is called in the source chains cancel() function. Source chain cancellations for cross chain swaps can only be cancelled by whitelisted solvers after the order has been expired, offerers are not permitted to cancel from the source chain at all.

3.5. An attacker can cancel any order

Target	Aori		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `cancel` does not verify that the `orderToCancel` matches the `orderId`, which should be the hash of an order. As a result, an attacker can use a malicious `orderToCancel` to bypass checks and cancel any order with an intended `orderId`.

```
function cancel(
    bytes32 orderId,
    Order calldata orderToCancel,
    bytes calldata extraOptions
) external payable nonReentrant whenNotPaused {
    orderToCancel.validateCancel(
        orderId,
        // [...]
    );
    // [...]
}

function validateCancel(
    IAori.Order calldata order,
    bytes32 orderId,
    // [...]
) internal view {
    require(order.dstEid == endpointId, "Not on destination chain");
    require(orderStatus(orderId) == IAori.OrderStatus.Unknown, "Order not
    active");
    require(
        (isAllowedSolver(sender)) ||
            (sender == order.offerer && block.timestamp > order.endTime),
        "Only whitelisted solver or offerer(after expiry) can cancel"
    );
}
```

Impact

The attacker can cancel an order in `orderToCancel.srcEid` by utilizing LayerZero's `lzSend` or can update an order status in this contract to prevent it from being used normally.

Recommendations

Consider checking whether the `orderId` and `orderToCancel` match.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `aadb3b4b` ↗.

Aori provided the following response to this finding:

> Cancel now requires the orderId submitted matches the derived orderHash from the submitted orderToCancel struct.

3.6. Unchecked return value on overflow in `increaseUnlockedNoRevert`

Target	Aori		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `increaseUnlockedNoRevert()` function handles overflow conditions by returning `false` instead of reverting execution. However, the boolean return value — indicating the failure condition — is not checked by the calling code in both the `depositAndFill()` and `_settleSingleChainSwap()` functions.

As a result, the `inputToken` balance of the solver may be handled incorrectly.

```
function increaseUnlockedNoRevert(
    Balance storage balance,
    uint128 amount
) internal returns (bool success) {
    uint128 unlocked = balance.unlocked;
    unchecked {
        uint128 newUnlocked = unlocked + amount;
        if (newUnlocked < unlocked) {
            return false; // Overflow
        }
        balance.unlocked = newUnlocked;
    }
    return true;
}

function depositAndFill(
    Order calldata order,
    bytes calldata signature
) external payable nonReentrant whenNotPaused onlySolver {
    [...]
    // Credit the input token directly to the solver's unlocked balance
    balances[msg.sender][order.inputToken].increaseUnlockedNoRevert(
        uint128(order.inputAmount));
    [...]
}

function _settleSingleChainSwap(
```

```
    bytes32 orderId,
    Order memory order,
    address solver,
    uint256 amountReceived
) internal {
    [...]
    if (balances[order.offerer][order.inputToken].locked >= order.inputAmount)
    {
        // Unlock the tokens from offerer's balance
        balances[order.offerer][order.inputToken].decreaseLockedNoRevert(
            uint128(order.inputAmount)
        );
        // Credit the tokens directly to the solver's unlocked balance
        balances[solver][order.inputToken].increaseUnlockedNoRevert(
            uint128(order.inputAmount));
    }
    [...]
}
```

Impact

If an overflow occurs and the returned `false` value is ignored, the contract may update the solver balance with an overflowed amount, leading to a loss of funds.

Recommendations

Check the boolean return value of the `increaseUnlockedNoRevert()` function in both `depositAndFill()` and `_settleSingleChainSwap()`. If the function returns `false`, revert the transaction.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit b4597419 ↗.

Aori provided the following response to this finding:

> Validatied balance change observations with a boolean successLock ensuring expected balance changes and transfer operation outcomes.

3.7. Incorrect transfer amount in the function `fill` with a hook

Target	Aori		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When using the function `fill` with a hook to fill an order, the function will observe the contract's output-token balance changes before and after the execution in the hook. If the amount of the output token received by the contract, `balChg` / `amountReceived`, is greater than the `order.outputAmount`, the excess output tokens will be sent to the solver. However, after transferring the surplus to the solver, the function transfers the full `amountReceived` (`balChg`) to the `order.recipient`, which means that the excess is also transferred to the recipient if the contract has sufficient output tokens.

```
function fill(
    Order calldata order,
    IAori.DstHook calldata hook
) external payable nonReentrant whenNotPaused onlySolver {
    // [...]
    uint256 amountReceived = _executeDstHook(order, hook);

    IERC20(order.outputToken).safeTransfer(order.recipient, amountReceived);
    _postFill(orderId, order);
}

function _executeDstHook(
    Order calldata order,
    IAori.DstHook calldata hook
) internal allowedHookAddress(hook.hookAddress) returns (uint256 balChg) {
    // [...]
    balChg = ExecutionUtils.observeBalChg(
        hook.hookAddress,
        hook.instructions,
        order.outputToken
    );
    require(balChg >= order.outputAmount, "Hook must provide at least the
    expected output amount");

    uint256 solverReturnAmt = balChg - order.outputAmount;
```

```
    if (solverReturnAmt > 0) {
        IERC20(order.outputToken).safeTransfer(msg.sender, solverReturnAmt);
    }
}
```

Impact

This can be used to drain tokens from the contract.

Recommendations

Transfer the exact `order.outputAmount` of `order.outputToken` to the `order.recipient`.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `66df7f90` ↗.

Aori provided the following response to this finding:

> Moved operations pertaining to surplus calculation and transfer to solver into fill() function for simplicity and made logic corrections.

3.8. Lack of validation in `deposit` that `dstEid` is supported

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

Description

The `deposit` function allows a solver to deposit tokens into the contract using the user's signature and the order's data. The order data includes a `dstEid` field, which specifies the destination chain where the filler should provide funds to the recipient to fulfill the order. However, the `deposit` function does not validate whether the provided `dstEid` is supported.

Impact

A deposit can be made even for unsupported chains. As a result, such orders will be impossible to fulfill and will eventually need to be canceled. This forces a solver to process meaningless operations and leads to unnecessary gas waste.

Recommendations

Add a validation check in the `deposit` function to ensure that the provided `dstEid` is supported before accepting the deposit.

Remediation

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

- 3492f209 ↗
- 99a623ac ↗

Aori provided the following response to this finding:

> Added an on-chain validation system for supported LayerZero endpoint IDs to prevent orders with unsupported destinations from being accepted. The solution adds a mapping to track valid chains, with admin functions addSupportedChain and removeSupportedChain. As well as addSupportedChains that allow the owner to add multiple chains. Added a destination validation check in validateDeposit that requires the destination chain to be supported before accepting deposits. The local chain is automatically marked as supported in the constructor.

3.9. Lack of validation in `_handleSettlement` that `order.dstEid` matches the sender `chainId`

Target	Aori		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

During settlement-message processing on the source chain, the `lzReceive` function is executed by the trusted LayerZero Endpoint contract. This function validates that the message was received from a trusted peer and that the expected sender for the given chain matches the actual sender. After these checks, the `_handleSettlement` function is called to settle all orders contained within the message.

However, the `_handleSettlement` function does not validate that each `order.dstEid`, where an order should be filled, actually matches the sender chain's `Eid`.

Impact

On the source chain, orders could potentially be settled if a message is received from any supported `dstEid` chain, even if the message was not sent from the exact chain that matches the specific `order.dstEid`.

However, the practical impact is informational, because the logic on the destination chain, which handles filling and settlement-message creation, already ensures that orders can only be filled and settled from the correct `dstEid`. Therefore, no direct exploit or incorrect settlement is possible under normal conditions, but this gap could weaken assumptions about message integrity if the system design changes in the future.

Recommendations

Add an additional validation step to ensure that the sender's `Eid` matches the `order.dstEid` before proceeding with settlement.

Remediation

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

- a8c00d9d ↗

- 31585f53 ↗

Aori provided the following response to this finding:

> The eid validation now being handled in _recvPayload by skipping settlements for orders that do not pass the validation check. In the case an order is skipped, a settlementFailed event is emitted containing the orderId.

3.10. Users are unable to cancel their own expired single-chain swap orders

Target	Aori		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

There are two types of supported cancellation:

1. On the source chain, the solver can cancel any active order.
2. On the destination chain, the solver or the offerer of an expired order can initiate cancellation-message sending. This message is then received and processed on the source chain to cancel the active order.

```
function cancel(bytes32 orderId) external whenNotPaused {
    require(
        isAllowedSolver[msg.sender],
        "Only whitelisted solver can cancel from the source chain"
    );
    _cancel(orderId);
}

function _cancel(bytes32 orderId) internal {
    require(orderStatus[orderId] == IAori.OrderStatus.Active, "Can only cancel
    active orders");
    orderStatus[orderId] = IAori.OrderStatus.Cancelled;
    [...]
}
function cancel(
    bytes32 orderId,
    Order calldata orderToCancel,
    bytes calldata extraOptions
) external payable nonReentrant whenNotPaused {
    orderToCancel.validateCancel(
        [...]
    );
    bytes memory payload = PayloadPackUtils.packCancellation(orderId);
    __lzSend(orderToCancel.srcEid, payload, extraOptions);
    orderStatus[orderId] = IAori.OrderStatus.Cancelled;
    emit CancelSent(orderId);
```

```
}

function validateCancel(
    [...]
) internal view {
    require(order.dstEid == endpointId, "Not on destination chain");
    require(orderStatus(orderId) == IAori.OrderStatus.Unknown, "Order not
    active");
    require(
        (isAllowedSolver(sender)) ||
            (sender == order.offerer && block.timestamp > order.endTime),
        "Only whitelisted solver or offerer(after expiry) can cancel"
    );
}
```

Impact

As a result, the offerer of an expired single-chain swap order can only cancel their order by either

- sending a cancellation message from the destination chain, or
- initiating a cancellation through the solver on the source chain.

However, single-chain swap orders are fully handled on the source chain only; their status is never updated on any destination chain. Therefore, these types of orders, if they have already expired, could be safely canceled directly by the offerer on the source chain without requiring cross-chain messaging or solver intervention.

Recommendations

Consider adding functionality on the source chain that allows the offerer to directly cancel expired single-chain swap orders without requiring a cross-chain cancellation process.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `113654cc` ↗.

Aori provided the following response to this finding:

> Whitelisted solvers can now cancel a single chain swap order that has not yet been filled. (This can only arise in the presumably rare case that a solver decides to delay fulfillment instead of choosing to fill atomically from inventory or with a hook.) The offerer can only cancel their own orders after the order has expired. Tokens are immediately unlocked and available for withdrawal by the offerer

3.11. Unsafe casting in `_postDeposit`

Target	Aori		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

There is unsafe typecasting in the `_postDeposit` function, where a `uint256 depositAmount` is cast to a `uint128`.

```
function _postDeposit(
    address depositToken,
    uint256 depositAmount,
    Order calldata order,
    bytes32 orderId
) internal {
    [...]
    orders[orderId].inputAmount = uint128(depositAmount);
    [...]
}
```

During the execution of the `deposit` function with a hook, if the `amountReceived` value returned from `_executeSrcHook` exceeds the `uint128` limit, the `_postDeposit` function will silently truncate the value to fit into `uint128`.

Impact

If the returned `amountReceived` value exceeds `uint128` limits and truncation occurs, the offerer or solver might receive significantly fewer tokens than expected when the order is canceled or settled.

Recommendations

It is recommended to use OpenZeppelin's SafeCast ↗ library for typecasting.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit b004bf6e ↗.

Aori provided the following response to this finding:

> Added use of OpenZeppelin's SafeCast library.

3.12. Unsafe memory manipulation

Target	Aori		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the function `packCancellation`, before calling the function `mstore`, the return variable `payload` is an unassigned empty bytes string, whose memory location is `0x60`.

```
function packCancellation(bytes32 orderHash)
    internal pure returns (bytes memory payload) {
    uint8 msgType = uint8(PayloadType.Cancellation);
    assembly {
        mstore(payload, 33)
        mstore8(add(payload, 32), msgType)
        mstore(add(payload, 33), orderHash)
        mstore(0x40, add(payload, 65))
    }
}
```

As a result, the assembly block overwrites the memory value at [`0x40` - `0xa1`], which contains the free memory pointer (`0x40` - `0x5f`) and the zero slot (`0x60` - `0x7f`).

Impact

According to the Solidity documentation ↗, the zero slot is used as an initial value for dynamic memory arrays and should never be written to. Overwriting this slot could potentially impact other parts of the code logic, although such effects are not observed in this contract.

Recommendations

It is recommended to use `abi.encodePacked` to pack the payload.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `dde53bf3` ↗.

Aori provided the following response to this finding:

> Gas savings of assembly optimization showed to be minimal, converted the function to safely use abi.encodePacked

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. The assembly block does not sanitize the high-order bits of the variable `locked`

The function `loadBalance` reads an entire 256-bit storage slot and returns this full value directly as the output parameter `locked`. It does not mask or clear the upper 128 bits before returning.

```
function loadBalance(
    Balance storage balance
) internal view returns (uint128 locked, uint128 unlocked) {
    assembly {
        let fullSlot := sload(balance.slot)
        unlocked := shr(128, fullSlot)
        locked := fullSlot
    }
}
```

Since the contract is compiled with `via_ir` set to true, it is not affected by this issue. However, when `via_ir` is false, this unsanitized `locked` value could trigger the incorrect calculation in the function `storeBalance`, resulting in an incorrect `unlocked` value being stored.

```
function storeBalance(Balance storage balance, uint128 locked,
    uint128 unlocked) internal {
    assembly {
        sstore(balance.slot, or(shl(128, unlocked), locked))
    }
}
```

Please ensure that `via_ir` is always set to true, or sanitize the upper 128 bits of the `locked` variable before returning in the function `loadBalance`.

This issue has been acknowledged by Aori.

4.2. The hook chosen by the solver may oppose user intent

There are two types of deposit functions; one requires a hook call and the other does not, but both need to be called by the solver. The solver decides whether to deposit an order with or without a hook.

The deposit with a hook requires the caller to provide the hook information, which is not included in the signed order. In some cases, the hook chosen by the solver might go against the user's intent. For a cross-chain deposit, the user will get the locked `hook.preferredToken`. If the user wants to cancel the order from the source chain, they can only receive `hook.preferredToken` instead of `order.inputToken`. If the `hook.preferredToken` is a nonstable token (e.g., WETH), it could potentially lead to user losses.

```
struct SrcHook {
    address hookAddress;
    address preferredToken;
    uint256 minPreferedTokenAmountOut;
    bytes instructions;
}
```

Consider providing a detailed explanation of this behavior in the documentation.

This issue has been acknowledged by Aori.

4.3. Unnecessary `payable` modifiers in functions

Several functions — such as `deposit`, `fill`, and `depositAndFill` — are marked as `payable`, but they do not process, validate, or utilize the received native tokens (`msg.value`) in any way. The native tokens sent to these functions are effectively ignored.

Since the `payable` modifier implies that the function is intended to receive and manage native tokens, its presence is misleading and may be safely removed.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit 638b1696 ↗.

Aori provided the following response to this finding:

> Removed unnecessary payable modifiers. Native token support will be implemented in a future release.

4.4. The `MessagingReceipt` returned from `endpoint.send` is ignored

The `settle` and `cancel` functions invoke `endpoint.send()` to initiate a cross-chain message via the LayerZero protocol, which returns a `MessagingReceipt` struct. This struct includes critical metadata such as the `guid` (unique identifier of the message), the `nonce` (the nonce of the message), and the `fee` (the LayerZero fee charged for sending the message). However, the contract currently ignores the returned `MessagingReceipt` entirely.

Including `MessagingReceipt` info as part of the `CancelSent` and `SettleSent` events would make it easier to trace cross-chain messages.

Remediation

[FIX: f36f04bdb9398c946ae420cebd366f84a917cc56]

Aori provided the following response to this finding:

> Included MessagingReceipt details with cancelSent and settleSent events.

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.

5.1. Module: Aori.sol

Function: `cancel(byte[32] orderId, Order orderToCancel, bytes extraOptions)`

This function allows a caller to send a cancellation message to the source chain. The caller can be a trusted solver or offerer of the expired order.

Inputs

- `orderId`
  - **Control**: Full control.
  - **Constraints**: The status of the order should be `Unknown`.
  - **Impact**: The ID of the order to be canceled.
- `orderToCancel`
  - **Control**: Full control.
  - **Constraints**: There is no verification that `orderId` actually matches the hash of the `orderToCancel`.
  - **Impact**: Contains information about the order to be canceled.
- `extraOptions`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Contains additional message configuration, including the gas amount and `msg.value` that the executor will use during message delivery.

Branches and code coverage

Intended branches

- The cancellation message is sent successfully.
  - ☐ Test coverage
- The `orderStatus` is updated to `Cancelled`.

- ☐ Test coverage

Negative behavior

- The `orderId` does not match the hash of the `orderToCancel`.
  - ☐ Negative test
- The caller is not a trusted solver or the offerer of the expired order.
  - ☐ Negative test
- The caller is an offerer of the order, but the order has not expired yet.
  - ☐ Negative test
- The `order.dstEid` != `endpointId`.
  - ☐ Negative test
- The status != `Unknown`.
  - ☐ Negative test

Function call analysis

- `orderToCancel.validateCancel(orderId, ENDPOINT_ID, this.orderStatus, msg.sender, this.isAllowedSolver);`
  - **What is controllable?** `orderId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function does not return a value.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the `order.dstEid != endpointId`, the `orderStatus(orderId) != IAori.OrderStatus.Unknown`, the caller is not a trusted solver, or the caller is not an offerer of the expired order.
- `PayloadPackUtils.packCancellation(orderId);`
  - **What is controllable?** `orderId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the packed cancellation payload — contains the `msgType` and `orderId`.
  - **What happens if it reverts, reenters or does other unusual control flow?** There are no problems here.
- `__lzSend(orderToCancel.srcEid, payload, extraOptions) -> _lzSend(eId, payload, extraOptions, MessagingFee(msg.value, 0), payable(msg.sender)) -> this.endpoint.send{value: messageValue}`
  - **What is controllable?** `messageValue`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `MessagingReceipt` data, which contains the unique identifier for the sent message; the nonce of the sent message; and the LayerZero fee

incurred for the message. But this data is ignored.
- **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the provided native tokens are not enough for the fee.

Function: `cancel(byte[32] orderId)`

This function allows a trusted solver to cancel any active order.

Inputs

- `orderId`
  - **Control**: Full control.
  - **Constraints**: The order should have `Active` status.
  - **Impact**: The order ID to cancel.

Branches and code coverage

Intended branches

- The active order is successfully cancelled.
  - ☐ Test coverage
- The status of the order is updated to `Cancelled`.
  - ☐ Test coverage
- The unlocked balances of the `order.offerer` for the `order.inputToken` are updated as expected.
  - ☐ Test coverage

Negative behavior

- The order is not `Active`.
  - ☐ Negative test
- The caller is not a trusted solver.
  - ☐ Negative test

Function call analysis

- `this._cancel(orderId) -> BalanceUtils.unlock(this.balances[order.offerer][order.inputToken], uint128(order.inputAmount))`

- **What is controllable?** N/A.
- **If the return value is controllable, how is it used and how can it go wrong?** This function does not return a value.
- **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the status of the order is not `Active`. Reverts if the `locked` balance of the `offerer` is less than the `inputAmount`.

Function: `depositAndFill(Order order, bytes signature)`

This function allows a solver to execute a single-chain order when the contract is not paused. After the deposit, the order will be filled immediately, and the final status of the order will be `Settled`. This function does not support tokens with features like fee-on-transfer or rebase.

Inputs

- `order`
  - **Control**: Fully controlled by the user.
  - **Constraints**: The order must be a single-chain order, and its status must be `Unknown`. The order information must meet some basic conditions — for example, the `block.timestamp` must be between the `order.startTime` and `order.endTime`, both `order.offerer` and `order.recipient` must not be zero address, and so on.
  - **Impact**: The order to be executed.
- `signature`
  - **Control**: Fully controlled by the user.
  - **Constraints**: It must be a valid signature signed by the `order.offerer`.
  - **Impact**: The user's EIP-712 signature over the order.

Branches and code coverage

Intended branches

- This function executes successfully.
  - ☑ Test coverage
- The final order status must be settled.
  - ☑ Test coverage
- The offerer's locked and unlocked balances must be unchanged.
  - ☑ Test coverage
- The solver's unlocked balance must be increased by the input amount.

☑ Test coverage

- The recipient's balance must be increased by the output amount.

☑ Test coverage

Negative behavior

- Reverts if it is a cross-chain order.

☑ Negative test

- Reverts if the order is expired.

☐ Negative test

- Reverts if the signature is invalid.

☑ Negative test

- Reverts if the order status is `Settled`.

☑ Negative test

- Reverts if the caller is not the solver.

☑ Negative test

- Reverts if the contract is paused.

☑ Negative test

- Reverts if this function is reentered.

☐ Negative test

Function call analysis

- `order.validateDepositAndFill(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus)`
  - **What is controllable?** `order` and `signature`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the `offerer`, `recipient`, `inputToken`, or `outputToken` address is zero; if `startTime >= endTime`; if `startTime` is in the future; if `endTime` is in the past; if `inputAmount` or `outputAmount` is zero; if the order status is not `Unknown`; if the signature is invalid; if `order.srcEid` or `order.dstEid` is not equal to `ENDPOINT_ID`; or if `inputToken == outputToken`.
- `IERC20(order.inputToken).safeTransferFrom(order.offerer, address(this), order.inputAmount)`
  - **What is controllable?** `order.inputToken`, `order.offerer`, and `order.inputAmount`.

- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the allowance of the input token granted to the contract by the offerer is insufficient to transfer the `inputAmount`.

- `IERC20(order.outputToken).safeTransferFrom(msg.sender, order.recipient, order.outputAmount)`
  - **What is controllable?** `order.outputToken`, `msg.sender`, `order.recipient`, and `order.outputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the allowance of the output token granted to the contract by the solver is insufficient to transfer the `outputAmount`. The `outputAmount` is transferred to the recipient.
- `balances[msg.sender][order.inputToken].increaseUnlockedNoRevert( uint128(order.inputAmount))`
  - **What is controllable?** `msg.sender`, `order.inputToken`, and `order.inputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value indicating whether the operation was successful is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** Adds the input amount to the solver's input-token unlocked balance. Integer overflow could happen.

Function: `deposit(Order order, bytes signature, SrcHook hook)`

This function allows a solver to perform a deposit with a `hook` call using an order with an EIP-712 signature previously signed by the user.

The order data contains `uint128 inputAmount`, `uint128 outputAmount`, `address inputToken`, `address outputToken`, `uint32 startTime`, `uint32 endTime`, `uint32 srcEid`, `uint32 dstEid`, `address offerer`, and `address recipient`.

The specified `inputAmount` of the `inputToken` is transferred from the `offerer` address to this contract. The `balances` mapping for the `offerer` and `inputToken` is increased by this `inputAmount`. The status of the order is set up to `Active`.

Inputs

- `order`
  - **Control**: Full control.

    - **Constraints**: The `order.srcEid` should be equal to the current `endpointId`. The current status of the order should be `Unknown`.
    - **Impact**: Contains information about the order.
- `signature`
    - **Control**: Full control.
    - **Constraints**: The signature should be valid for the `order.offerer` address.
    - **Impact**: The EIP-712 signature of the provided `order` data.
- `hook`
    - **Control**: Full control.
    - **Constraints**: Only `allowedHookAddress` is acceptable.
    - **Impact**: Contains the `hookAddress`, `preferredToken address`, `minPreferedTokenAmountOut`, and additional instructions for the hook contract.

Branches and code coverage

Intended branches

- The tokens are provided successfully.
    - ☐ Test coverage
- `balances[order.offerer][depositToken]` is updated by the `inputAmount`.
    - ☐ Test coverage
- `orderStatus` is updated to `Active`.
    - ☐ Test coverage
- In the case of `isSingleChainSwap`, the `offerer` receives the expected `outputToken` and specified `outputAmount`.
    - ☐ Test coverage
- Otherwise, the `order` data is updated, the `outputToken` is updated to the `hook.preferredToken` address, and the specified `outputAmount` is changed to the `amountReceived`.
    - ☐ Test coverage

Negative behavior

- The caller is not the solver.
    - ☐ Negative test
- Invalid signature is used.
    - ☐ Negative test
- The signature has been used already by the solver.

- ☐ Negative test

- `order.srcEid != endpointId`.
  - ☐ Negative test
- `order.offerer == address(0)`.
  - ☐ Negative test
- `order.recipient == address(0)`.
  - ☐ Negative test
- `order.startTime >= order.endTime`.
  - ☐ Negative test
- `order.startTime > block.timestamp`.
  - ☐ Negative test
- `order.endTime < block.timestamp`.
  - ☐ Negative test
- `order.inputAmount == 0`.
  - ☐ Negative test
- `order.outputAmount == 0`.
  - ☐ Negative test
- `order.inputToken == address(0)`.
  - ☐ Negative test
- `order.outputToken == address(0)`.
  - ☐ Negative test
- The `hook` is not allowed.
  - ☐ Negative test
- The `amountReceived` after `hook` execution is less than `order.outputAmount` if `isSingleChainSwap`.
  - ☐ Negative test
- Otherwise, `amountReceived` after `hook` execution is less than `hook.minPreferedTokenAmountOut`.
  - ☐ Negative test

Function call analysis

- `hook.isSome()`
  - **What is controllable?** `hook`.

    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the bool result of comparing the `hook.hookAddress` with zero address.
    - **What happens if it reverts, reenters or does other unusual control flow?** Transaction will revert if the returned result is false; it means that the solver did not provide the nonzero `hookAddress`.
- `_hashOrder712(order)`
    - **What is controllable?** `order`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the EIP-712 digest, which is used for recovering the signer's address from the provided signature.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.
- `order.validateDeposit(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus) -> keccak256(abi.encode(order))`
    - **What is controllable?** `order`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `orderId`, computed as the hash of the given order struct. The `orderId` is used as the unique ID of the order, and further it is used to verify that this order has not been deposited yet, to avoid a replay attack.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.
- `order.validateDeposit(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus) -> orderStatus(orderId)`
    - **What is controllable?** `signature` and `order`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current status of the `orderId`.
    - **What happens if it reverts, reenters or does other unusual control flow?** If the returned order status is not `Unknown`, the transaction will be reverted.
- `order.validateDeposit(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus) -> ECDSA.recover(digest, signature)`
    - **What is controllable?** `digest` and `signature`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the address of the signer; it should match the `offerer` address from the `order` data.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.
- `order.validateDeposit(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus) -> validateCommonOrderParams(order)`
    - **What is controllable?** `order`.
    - **If the return value is controllable, how is it used and how can it go wrong?**

There is no return value here.

- **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the `offerer`, `recipient`, `inputToken`, or `outputToken` address is zero; if `startTime >= endTime`; if `startTime` is in the future; if `endTime` is in the past; or if `inputAmount` or `outputAmount` is zero.

- `_executeSrcHook(order, hook) -> IERC20(order.inputToken).safeTransferFrom(order.offerer, hook.hookAddress, order.inputAmount)`
  - **What is controllable?** `order.offerer`, `hook.hookAddress`, and `order.inputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the allowance of `inputToken` granted to the contract by the `offerer` is insufficient to transfer the `inputAmount`. The `inputAmount` is transferred to the `hookAddress` instead of the current contract address.
- `_executeSrcHook(order, hook) -> order.isSingleChainSwap()`
  - **What is controllable?** `order`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true if `srcEid` matches `dstEid`.
  - **What happens if it reverts, reenters or does other unusual control flow?** There are no problems here.
- `_executeSrcHook(order, hook) -> ExecutionUtils.observeBalChg(hook.hookAddress, hook.instructions, order.outputToken)`
  - **What is controllable?** `hook.hookAddress`, `hook.instructions`, and `order.outputToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the amount of the `order.outputToken` received by this contract as a result of the `hook` execution. If this amount is less than the `order.outputAmount`, the transaction will be reverted. But this is only for the case `isSingleChainSwap`.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the hook call has failed. Also, it reverts if the balance of `outputToken` of this contract after the hook call is less than the balance before.
- `_executeSrcHook(order, hook) -> ExecutionUtils.observeBalChg(hook.hookAddress, hook.instructions, hook.preferredToken)`
  - **What is controllable?** `hook.hookAddress`, `hook.instructions`, and `hook.preferredToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?**

Returns the amount of the `hook.preferredToken`. The `preferredToken` address is controlled by the solver, received by this contract as a result of the `hook` execution. If the `amountReceived` is less than the `hook.minPreferedTokenAmountOut` (also controlled by the solver), the transaction will be reverted.

- **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the hook call has failed. Also, it reverts if the balance of `preferredToken` of this contract after the hook call is less than the balance before.

- `order.isSingleChainSwap()`
  - **What is controllable?** `order`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true if `srcEid` matches `dstEid`.
  - **What happens if it reverts, reenters or does other unusual control flow?** There are no problems here.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> IERC20(order.outputToken).safeTransfer(order.recipient, order.outputAmount)`
  - **What is controllable?** `order.outputToken`, `order.recipient`, and `order.outputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Transfers the original `order.outputAmount` of the `order.outputToken` to the `order.recipient`.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> IERC20(order.outputToken).safeTransfer(solver, surplus)`
  - **What is controllable?** `order.outputToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Transfers the `surplus` of the `order.outputToken` to the caller.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> balances[order.offerer][order.inputToken].decreaseLockedNoRevert( uint128(order.inputAmount))`
  - **What is controllable?** `order.offerer`, `order.inputToken`, and `order.inputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns `false` if the `offerer` balance has been underflowed as a result of decreasing by the specified `amount`. But the result of this function is ignored. This is not an issue here since changes to the locked balance are checked after this.

    - **What happens if it reverts, reenters or does other unusual control flow?** This balance update occurs after the `offerer` has already provided funds by direct transfer in `_settleSingleChainSwap`, which actually leads to a double debit of tokens.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> balances[solver][order.inputToken].increaseUnlockedNoRevert( uint128(order.inputAmount))`
    - **What is controllable?** `order.inputToken` and `order.inputAmount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns `false` if the `solver` balance has been overflowed as a result of increasing by the specified `amount`. But the result of this function is ignored, and as a result, if the balance has been overflowed, funds will be lost.
    - **What happens if it reverts, reenters or does other unusual control flow?** If the `locked` balance of the `offerer` is insufficient, the `solver`'s `unlocked` balance will not be increased; as a result, the transaction will be reverted since the `balances` verification is at the end of the `_settleSingleChainSwap` function.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> balances[order.offerer][order.inputToken].validateBalanceTransferOrRevert( initialOffererLocked, finalOffererLocked, initialSolverUnlocked, finalSolverUnlocked, uint128(order.inputAmount))`
    - **What is controllable?** `order.inputAmount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the resulting balance of the solver is not equal to the initial balance plus `inputAmount` and if the initial balance of the offerer is not equal to the resulting balance plus `inputAmount`.
- `_postDeposit(tokenReceived, amountReceived, order, orderId)`
    - **What is controllable?** `order`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
    - **What happens if it reverts, reenters or does other unusual control flow?** This function rewrites the initial `order.inputToken` by the `hook.preferredToken` address and the `order.inputAmount` by the `amountReceived` received as a result of the `hook` execution.

Function: `deposit(Order order, bytes signature)`

This function allows the solver to perform a deposit using an order with an EIP-712 signature previously signed by the user.

Order data contains `uint128 inputAmount`, `uint128 outputAmount`, `address inputToken`,

`address outputToken`, `uint32 startTime`, `uint32 endTime`, `uint32 srcEid`, `uint32 dstEid`, `address offerer`, and `address recipient`.

The specified `inputAmount` of the `inputToken` is transferred from the `offerer` address to this contract. The `balances` mapping for the `offerer` and `inputToken` is increased by this `inputAmount`. The status of the order is set up to `Active`.

Inputs

- `order`
  - **Control**: Full control.
  - **Constraints**: `order.srcEid` should be equal to the current `endpointId`. The current status of the order should be `Unknown`.
  - **Impact**: Contains information about the order.
- `signature`
  - **Control**: Full control.
  - **Constraints**: The signature should be valid for the `order.offerer` address.
  - **Impact**: The EIP-712 signature of the provided `order` data.

Branches and code coverage

Intended branches

- The tokens are provided successfully.
  - ☐ Test coverage
- `balances[order.offerer][depositToken]` is updated by the `inputAmount`.
  - ☐ Test coverage
- `orderStatus` is updated to `Active`.
  - ☐ Test coverage

Negative behavior

- Caller is not the solver.
  - ☐ Negative test
- Invalid signature is used.
  - ☐ Negative test
- The signature has been used already by the solver.
  - ☐ Negative test
- `order.srcEid != endpointId`.

- ☐ Negative test

- `order.offerer == address(0)`.
  - ☐ Negative test
- `order.recipient == address(0)`.
  - ☐ Negative test
- `order.startTime >= order.endTime`.
  - ☐ Negative test
- `order.startTime > block.timestamp`.
  - ☐ Negative test
- `order.endTime < block.timestamp`.
  - ☐ Negative test
- `order.inputAmount == 0`.
  - ☐ Negative test
- `order.outputAmount == 0`.
  - ☐ Negative test
- `order.inputToken == address(0)`.
  - ☐ Negative test
- `order.outputToken == address(0)`.
  - ☐ Negative test

Function call analysis

- `_hashOrder712(order)`
  - **What is controllable?** `order`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the EIP-712 digest, which is used for recovering the signer's address from the provided signature.
  - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.
- `order.validateDeposit(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus) -> keccak256(abi.encode(order))`
  - **What is controllable?** `order`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `orderId`, computed as the hash of the given order struct. The `orderId` is used as a unique ID of the order, and further it is used to verify that this order has not been deposited yet, to avoid a replay attack.

    - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.
- `order.validateDeposit(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus) -> orderStatus(orderId)`
    - **What is controllable?** `signature` and `order`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current status of the `orderId`.
    - **What happens if it reverts, reenters or does other unusual control flow?** If the returned order status is not `Unknown`, the transaction will be reverted.
- `order.validateDeposit(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus) -> ECDSA.recover(digest, signature)`
    - **What is controllable?** `digest` and `signature`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the address of the signer; it should match the `offerer` address from the `order` data.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.
- `order.validateDeposit(signature, _hashOrder712(order), ENDPOINT_ID, this.orderStatus) -> validateCommonOrderParams(order)`
    - **What is controllable?** `order`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the `offerer`, `recipient`, `inputToken`, or `outputToken` address is zero; if `startTime >= endTime`; if `startTime` is in the future; if `endTime` is in the past; or if `inputAmount` or `outputAmount` is zero.
- `IERC20(order.inputToken).safeTransferFrom(order.offerer, address(this), order.inputAmount)`
    - **What is controllable?** `order.inputToken`, `order.offerer`, and `order.inputAmount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the allowance of `inputToken` granted to the contract by the `offerer` is insufficient to transfer the `inputAmount`.
- `_postDeposit(order.inputToken, order.inputAmount, order, orderId)`
    - **What is controllable?** `order.inputToken`, `order.inputAmount`, and `order`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.

Function: `fill(Order order)`

This function allows a solver to fill a cross-chain or a single-chain order when the contract is not paused. The final status of the order will be `Filled` for a cross-chain order and `Settled` for a single-chain order. This function does not support tokens with features like fee-on-transfer or rebase.

Inputs

- `order`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `order.dstEid` must match the `ENDPOINT_ID`. The order status must be `Active` for single-chain swaps and `Unknown` for cross-chain swaps. The order information must meet some basic conditions — for example, the `block.timestamp` must be between the `order.startTime` and `order.endTime`, both `order.offerer` and `order.recipient` must not be zero address, and so on.
  - **Impact**: The order to be filled.

Branches and code coverage

Intended branches

- This function executes successfully.
  - ☑ Test coverage
- For a cross-chain order, the recipient's output-token balance must be increased by the output amount.
  - ☑ Test coverage
- For a single-chain order, the solver's output-token balance must be decreased by the output amount.
  - ☑ Test coverage
- For a single-chain order, the recipient's output-token balance must be increased by the output amount.
  - ☐ Test coverage
- For a single-chain order, the order's final status must be `Settled`.
  - ☑ Test coverage

Negative behavior

- Reverts if the solver cannot provide enough output tokens.
  - ☑ Negative test

- Reverts if the single-chain order status is not `Active`.
  - ☑ Negative test
- Reverts if the `order.inputAmount` is zero.
  - ☑ Negative test
- Reverts if the `order.outputAmount` is zero.
  - ☑ Negative test
- Reverts if the order has not started.
  - ☑ Negative test
- Reverts if the order is expired.
  - ☑ Negative test
- Reverts if the `order.dstEid` does not match the `ENDPOINT_ID`.
  - ☑ Negative test
- Reverts if the caller is not a solver.
  - ☑ Negative test
- Reverts if the contract is paused.
  - ☑ Negative test

Function call analysis

- `order.validateFill(ENDPOINT_ID, this.orderStatus)`
  - **What is controllable?** `order`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the `offerer`, `recipient`, `inputToken`, or `outputToken` address is zero; if `startTime >= endTime`; if `startTime` is in the future; if `endTime` is in the past; if `inputAmount` or `outputAmount` is zero; if `order.dstEid != ENDPOINT_ID`; if the single-chain order status is not `Active`; or if the cross-chain order status is not `Unknown`.
- `IERC20(order.outputToken).safeTransferFrom(msg.sender, order.recipient, order.outputAmount)`
  - **What is controllable?** `order.outputToken`, `msg.sender`, `order.recipient`, and `order.outputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the allowance of output token granted to the contract by the solver

is insufficient to transfer the `outputAmount`. The `outputAmount` is transferred to the recipient.

- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> IERC20(order.outputToken).safeTransfer(order.recipient, order.outputAmount)`
  - **What is controllable?** `order.outputToken`, `order.recipient`, and `order.outputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Transfers the original `order.outputAmount` of output token from the contract to the recipient.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> IERC20(order.outputToken).safeTransfer(solver, surplus)`
  - **What is controllable?** `order.outputToken` and `solver`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If there are a surplus of output tokens, it transfers them from the contract to the solver.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> balances[order.offerer][order.inputToken].decreaseLockedNoRevert( uint128(order.inputAmount))`
  - **What is controllable?** `order.offerer`, `order.inputToken`, and `order.inputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value indicating whether the operation was successful is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** Subtracts the `order.inputAmount` from the locked input token balance of the offerer using the assembly block. Integer underflow could happen.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> balances[solver][order.inputToken].increaseUnlockedNoRevert( uint128(order.inputAmount))`
  - **What is controllable?** `solver`, `order.inputToken`, and `order.inputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value indicating whether the operation was successful is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** Adds the `order.inputAmount` to the unlocked input token balance of the solver using the assembly block. Integer overflow could happen.
- `_settleSingleChainSwap(orderId, order, msg.sender, amountReceived) -> balances[order.offerer][order.inputToken].validateBalanceTransferOrRevert( initialOffererLocked, finalOffererLocked, initialSolverUnlocked,`

`finalSolverUnlocked, uint128(order.inputAmount))`

- **What is controllable?** `order.offerer`, `order.inputToken`, and `order.inputAmount`.
- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** Validates the balance changes and reverts if there is integer overflow/underflow.

- `_postFill(orderId, order)`
  - **What is controllable?** `orderId` and `order`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** For a cross-chain order, it updates the order status to `Filled` and pushes the `orderId` to the `srcEidToFillerFills` mapping.

Function: `fill(Order order, IAori.DstHook hook)`

This function allows a solver to fill a cross-chain order with a hook when the contract is not paused. The final status of the order will be `Filled`. This function supports output tokens with features like fee-on-transfer or rebase.

Inputs

- `order`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The order must be a cross-chain order, and its status must be `Unknown`. The `order.dstEid` must match the `ENDPOINT_ID`. The order information must meet some basic conditions — for example, the `block.timestamp` must be between the `order.startTime` and `order.endTime`, both `order.offerer` and `order.recipient` must not be zero address, and so on.
  - **Impact**: The order to be filled.
- `hook`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The hook address must be whitelisted.
  - **Impact**: The hook configuration to be executed.

Branches and code coverage

Intended branches

- This function executes successfully.
  - ☑ Test coverage
- The recipient's output token balance must be increased by the output amount when there is no surplus.
  - ☑ Test coverage
- The recipient's output token balance must be increased by the output amount when there is a surplus.
  - ☐ Test coverage

Negative behavior

- Reverts if the `order.startTime` is greater than the `order.endTime`.
  - ☑ Negative test
- Reverts if the `order.inputAmount` is zero.
  - ☑ Negative test
- Reverts if the `order.outputAmount` is zero.
  - ☑ Negative test
- Reverts if the order has not started.
  - ☑ Negative test
- Reverts if the order is expired.
  - ☑ Negative test
- Reverts if there is not enough output tokens after execution.
  - ☑ Negative test
- Reverts if the execution of the hook fails.
  - ☑ Negative test
- Reverts if the hook address is not whitelisted.
  - ☑ Negative test

Function call analysis

- `order.validateFill(ENDPOINT_ID, this.orderStatus)`
  - **What is controllable?** `order`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.

- **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the `offerer`, `recipient`, `inputToken`, or `outputToken` address is zero; if `startTime >= endTime`; if `startTime` is in the future; if `endTime` is in the past; if `inputAmount` or `outputAmount` is zero; if `order.dstEid != ENDPOINT_ID`; if the single-chain order status is not `Active`; or if the cross-chain order status is not `Unknown`.

- `_executeDstHook(order, hook) -> IERC20(hook.preferredToken).safeTransferFrom(msg.sender, hook.hookAddress, hook.preferedDstInputAmount)`
  - **What is controllable?** `hook.preferredToken`, `msg.sender`, `hook.hookAddress`, and `hook.preferedDstInputAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Transfers the preferred token from the solver to the hook when `msg.value` is zero and `hook.preferedDstInputAmount` is greater than zero. Reverts if the allowance of `hook.preferredToken` granted to the contract by the solver is insufficient to transfer the `hook.preferedDstInputAmount`.
- `_executeDstHook(order, hook) -> ExecutionUtils.observeBalChg(hook.hookAddress, hook.instructions, order.outputToken)`
  - **What is controllable?** `hook.hookAddress`, `hook.instructions`, and `order.outputToken`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the amount of the `order.outputToken`'s balance change before and after the execution on the hook.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the hook call has failed or if the amount of the `order.outputToken` received is less than the `order.outputAmount`.
- `_executeDstHook(order, hook) -> IERC20(order.outputToken).safeTransfer(msg.sender, solverReturnAmt)`
  - **What is controllable?** `order.outputToken`, `msg.sender`, and `solverReturnAmt`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Transfers the surplus output token from the contract to the solver.
- `_postFill(orderId, order)`
  - **What is controllable?** `order`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?**

Updates the order status to `Filled` and pushes the `orderId` to the `srcEidToFillerFills` mapping.

Function: `settle(uint32 srcEid, address filler, bytes extraOptions)`

This function allows any caller to send to the `srcEid` chain the settlement message containing the settled order's ID for the `srcEid` and `filler`.

Inputs

- `srcEid`
  - **Control**: Full control.
  - **Constraints**: The `srcEidToFillerFills[srcEid][filler]` should contain at least one element.
  - **Impact**: The `srcEid` of the chain the settlement message will be sent on.
- `filler`
  - **Control**: Full control.
  - **Constraints**: The `srcEidToFillerFills[srcEid][filler]` should contain at least one element.
  - **Impact**: The address of the filler of the orders.
- `extraOptions`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Contains additional message configuration, including the gas amount and `msg.value` that the executor will use during message delivery.

Branches and code coverage

Intended branches

- The settlement message is sent successfully and contains all expected `orderIds`.
  - ☑ Test coverage
- The length `srcEidToFillerFills[srcEid][filler]` exceeds the `MAX_FILLS_PER_SETTLE`, and the `srcEidToFillerFills` is updated properly to remove the orders only for the first `MAX_FILLS_PER_SETTLE` elements.
  - ☑ Test coverage

Negative behavior

- There are no filled orders for `srcEid` and `filler`.

- ☐ Negative test

- The `srcEid` is not supported.
  - ☐ Negative test

Function call analysis

- `PayloadPackUtils.packSettlement(arr, filler, fillCount)`
  - **What is controllable?** `filler`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns `payload`, which contains message type, filler address, fill count, and the `orderIds` from the `arr`.
  - **What happens if it reverts, reenters or does other unusual control flow?** No problems here.
- `this._lzSend(srcEid, payload, extraOptions, MessagingFee(msg.value, 0), address payable(msg.sender)) -> this._payLzToken(_fee.lzTokenFee) -> this.endpoint.lzToken()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the address of the `lzToken` from the LZ endpoint contract, but this address is not actually used since `MessagingFee.lzTokenFee` is set up to zero.
  - **What happens if it reverts, reenters or does other unusual control flow?** This code is not executed since `MessagingFee.lzTokenFee` is set up to zero.
- `this._lzSend(srcEid, payload, extraOptions, MessagingFee(msg.value, 0), address payable(msg.sender)) -> this._payLzToken(_fee.lzTokenFee) -> SafeERC20.safeTransferFrom(IERC20(lzToken), msg.sender, address(this.endpoint), _lzTokenFee)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** This code is not executed since `MessagingFee.lzTokenFee` is set up to zero.
- `this._lzSend(srcEid, payload, extraOptions, MessagingFee(msg.value, 0), address payable(msg.sender)) -> this.endpoint.send{value: messageValue}`
  - **What is controllable?** `messageValue`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `MessagingReceipt` data, which contains the unique identifier for the sent message, the nonce of the sent message, and the LayerZero fee incurred for the message. But this data is ignored.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the provided native tokens are not enough for the fee.

Function: `withdraw(address token)`

This function allows users to withdraw their unlocked token balances when the contract is not paused.

Inputs

- `token`
  - **Control**: Fully controlled by the user.
  - **Constraints**: The `balances[msg.sender][token].unlocked` must be greater than zero.
  - **Impact**: The token to be withdrawn.

Branches and code coverage

Intended branches

- This function executes successfully.
  - ☑ Test coverage
- The holder's unlocked token balance must be zero after withdrawal.
  - ☑ Test coverage
- The holder's token balance must increase by the unlocked amount.
  - ☑ Test coverage

Negative behavior

- Reverts if the contract is paused.
  - ☑ Negative test
- Reverts if the `balances[holder][token].unlocked` is zero.
  - ☐ Negative test

Function call analysis

- `IERC20(token).safeTransfer(holder, amount)`
  - **What is controllable?** `token` and `holder`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Transfers the unlocked token balance to the holder and reverts if the contract's token balance is insufficient.

Function: `_lzReceive(Origin origin, bytes32 guid, bytes payload, address executor, bytes extraData)`

This function handles incoming LayerZero messages for source-chain order settlement and cancellation when the contract is not paused.

Inputs

- `origin`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a whitelisted sender in the source endpoint.
  - **Impact**: The origin information containing the source endpoint and the sender address.
- `guid`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The unique identifier for the received LayerZero message. Unused in this function.
- `payload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The message type must be either `Cancellation` or `Settlement`. If the type is `Cancellation`, the payload length must be 33 bytes. If the type is `Settlement`, the payload length must be greater than 23 bytes.
  - **Impact**: The message payload containing order hashes and filler information.
- `executor`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The address of the executor for the received message. Unused in this function.
- `extraData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Additional arbitrary data provided by the corresponding executor. Unused in this function.

Branches and code coverage

Intended branches

- This function executes successfully.

☑ Test coverage

- The filler's unlocked input-token balance must be increased by the input amount for a settlement.

☑ Test coverage

- The order status must be updated to `Cancelled` for a cancellation.

☑ Test coverage

- The offerer's locked input token must be unlocked for a cancellation.

☑ Test coverage

Negative behavior

- Reverts if the payload length is invalid.

☑ Negative test

- Reverts if the sender is not whitelisted.

☑ Negative test

- Reverts if the endpoint is invalid.

☑ Negative test

- Reverts if the payload type is unsupported.

☑ Negative test

Function call analysis

- `_recvPayload(payload) -> _handleCancellation(payload) -> _cancel(orderId)`
  - **What is controllable?** `payload`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the `orderStatus[orderId]` is not `Active`.
- `_recvPayload(payload) -> _handleSettlement(payload) -> _settleOrder(orderId, filler)`
  - **What is controllable?** `payload`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the `orderStatus[orderId]` is not `Active` or if the locked token transfer fails, it will simply skip this order.

6. Assessment Results

During our assessment on the scoped Aori contracts, we discovered 12 findings. No critical issues were found. Three findings were of high impact, four were of medium impact, one was of low impact, and the remaining findings were informational in nature.

At the time of our assessment, the reviewed code was not yet deployed to Ethereum, Optimism, Arbitrum, or Base Mainnet.

There are two **key areas for enhancement**.

1. Test coverage for interactions with special ERC-20 tokens, such as fee-on-transfer and rebasing tokens, is limited.
2. When testing with transactions involving token transfers, it is important to verify the token-balance changes of the associated addresses.

Here are the **recommended next steps**.

- Expand test coverage for interactions with special ERC-20 tokens.
- Improve the observation of the balance changes in transactions involving token transfers.
- Deploy to the testnet to validate intended behavior under real conditions.

While the foundation is promising, implementing these recommendations would provide greater assurance for mainnet deployment.

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.