Zellic

Prepared for
Joshua Baker
Wyatt Cravatas
Aori

Prepared by
Katerina Belotskaia
Weipeng Lai
Zellic

June 24, 2025

Aori 0.3.1 Upgrade

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 June 19th to June 20th, 2025. During this engagement, Zellic reviewed Aori 0.3.1 Upgrade'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 vulnerabilities that could cause direct or indirect loss of user or protocol funds?
- Does native token support create opportunities for reentrancy attacks?
- Are there edge cases in balance tracking that could result in inconsistent states?

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.

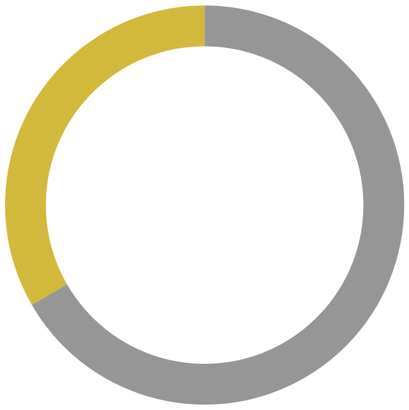
1.4. Results

During our assessment on the scoped Aori 0.3.1 Upgrade contracts, we discovered six findings. No critical issues were found. Two findings were of medium 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. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	2
🟩 Low	0
⬜ Informational	4



2. Introduction

2.1. About Aori 0.3.1 Upgrade

Aori contributed the following description of Aori 0.3.1 Upgrade:

> Aori is designed to securely facilitate performant trade execution with trust minimized settlement from any chain to any chain. To accomplish this, Aori uses a combination of off-chain infrastructure, on-chain settlement contracts, and Layer Zero messaging. Solvers expose a simple API to ingest and process order-flow directly to their trading system. The Aori smart contracts ensure that the user's intents are satisfied by the Solver on the destination chain according to the parameters of an intent on the source chain, signed by the user.

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 0.3.1 Upgrade Contracts

Type	Solidity
Platform	EVM-compatible
Target	Only the changes made from e0dceff8e294427f663c6787ed71c1378bd6d863 in the repository to the following
Repository	https://github.com/aori-io/aori ↗
Version	`1b3d4bb8c4076b6e3498c61c2ffc683859dd8c86`
Programs	`Aori.sol` `AoriUtils.sol`

2.4. Project Overview

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

Weipeng Lai
Engineer
weipeng.lai@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 19, 2025	Start of primary review period
June 20, 2025	End of primary review period

3. Detailed Findings

3.1. Missing reentrancy guard in the `cancel` function

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

Description

The source-chain `cancel` function allows a solver to cancel an active single-chain order or allows the offerer to cancel the single-chain order when it expires.

However, the `cancel` function lacks a `nonReentrant` modifier, exposing it to potential reentrancy attacks.

```
function cancel(bytes32 orderId) external whenNotPaused {
```

Impact

If a solver's account were compromised, an attacker could exploit this vulnerability for profit. By converting the solver's address into a malicious contract via EIP-7702, the attacker can execute a reentrancy attack during a native token transfer.

An attacker could perform the following steps:

1. **Create orders.** The attacker, in control of a solver on Arbitrum, creates two orders.
    - Order 1: Deposit 2,500 USDT on Arbitrum → receive 1 ETH on Arbitrum (recipient: solver address)
    - Order 2: Deposit 2,500 USDT on Arbitrum → receive 2,500 USDC on Optimism
2. **Fill order 1.** The attacker uses the compromised solver to fill order 1. The native ETH transfer to the solver's address triggers the malicious contract's fallback function.
3. **Reenter `cancel`.** Inside the fallback function, the malicious contract calls the cancel function for order 1. Because there is no reentrancy guard, the call succeeds. The attacker's locked USDT balance decreases due to both cancel and fill actions, resulting in a zero locked balance.
4. **Exploit order 2.** Since the attacker's locked USDT balance on the source chain (Arbitrum) is zero, any attempt to settle order 2 will fail on Arbitrum. However, solvers on the destination chain (Optimism) may still attempt to fill order 2. Each time an Optimism solver fills the order, the attacker receives 2,500 USDC on Optimism, while the order status remains active because the source-chain settlement fails repeatedly.

Recommendations

Add a reentrancy guard to the `cancel` function.

```
function cancel(bytes32 orderId) external whenNotPaused {
function cancel(bytes32 orderId) external nonReentrant whenNotPaused {
```

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `0b18c84f` ↗.

3.2. Incorrect state update on settlement failure

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

Description

The `_settleOrder` function settles a single order and updates balances. When balance updates fail, it skips settling the current order by returning early.

```
function _settleOrder(bytes32 orderId, address filler) internal {
    // [...]
    bool successLock
    = balances[order.offerer][order.inputToken].decreaseLockedNoRevert(
        uint128(order.inputAmount)
    );
    bool successUnlock
    = balances[filler][order.inputToken].increaseUnlockedNoRevert(
        uint128(order.inputAmount)
    );

    if (!successLock || !successUnlock) {
        return; // Any reverts are skipped
    }
    // [...]
}
```

However, an incorrect balance state could occur in this scenario — if `successLock` is false but `successUnlock` is true, the function returns after increasing the filler's unlocked balance, creating an inconsistent state where

- the filler's unlocked amount increases,
- the offerer's locked amount remains unchanged, and
- the order status remains unchanged.

Impact

Incorrect state updates on settlement failure in the `_settleOrder` function could result in inconsistent balance accounting.

Recommendations

We recommend restoring the original balances when either operation fails:

```
function _settleOrder(bytes32 orderId, address filler) internal {
    if (orderStatus[orderId] != IAori.OrderStatus.Active) {
        return; // Any reverts are skipped
    }
    // Update balances: move from locked to unlocked
    Order memory order = orders[orderId];
    Balance memory offererBalanceCache = balances[order.offerer][order.
        inputToken];
    Balance memory fillerBalanceCache = balances[filler][order.inputToken];
    bool successLock
    = balances[order.offerer][order.inputToken].decreaseLockedNoRevert(
        uint128(order.inputAmount)
    );
    bool successUnlock
    = balances[filler][order.inputToken].increaseUnlockedNoRevert(
        uint128(order.inputAmount)
    );

    if (!successLock || !successUnlock) {
        balances[order.offerer][order.inputToken] = offererBalanceCache;
        balances[filler][order.inputToken] = fillerBalanceCache;
        return; // Any reverts are skipped
    }
    orderStatus[orderId] = IAori.OrderStatus.Settled;

    emit Settle(orderId);
}
```

Remediation

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

- 6cc71403 ↗
- c0a6eb79 ↗

3.3. The `deposit` function with a hook supports native tokens

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

Description

The updated Aori contract introduces a new `depositNative` function designed exclusively for handling orders with a native token deposit. However, the `deposit` function with a hook still includes logic to process cases where the specified `inputToken` is the `NATIVE_TOKEN` address and assumes that the caller will supply the corresponding native token amount.

```
function deposit(
    Order calldata order,
    bytes calldata signature,
    SrcHook calldata hook
) external nonReentrant whenNotPaused onlySolver {
    [...]
    // Execute hook and handle single-chain or cross-chain logic
    (uint256 amountReceived, address tokenReceived) =
        _executeSrcHook(order, hook);
    [...]
}

function _executeSrcHook(
    Order calldata order,
    SrcHook calldata hook
) internal allowedHookAddress(hook.hookAddress) returns (
    uint256 amountReceived,
    address tokenReceived
) {
    // Transfer input tokens to the hook
    if (order.inputToken.isNativeToken()) {
        require(msg.value == order.inputAmount, "Incorrect native amount");
        (bool success, ) = payable(hook.hookAddress).call{value:
    order.inputAmount}("");
        require(success, "Native transfer to hook failed");
    [...]
}
```

Impact

Although the `deposit` function is not meant to handle native tokens, the `_executeSrcHook` function still allows them to be provided. The actual impact is informational since the `deposit` function is not marked as `payable`, which means native tokens cannot be sent during the call.

Recommendations

We recommend removing the native token–handling logic from the `_executeSrcHook` function, as this code path is unused and adds unnecessary complexity. Additionally, we suggest adding a check in the `deposit` function with a hook to ensure that `inputToken` cannot be a native token.

Remediation

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

- `ff76040c` ↗
- `c023f0cd` ↗

3.4. Redundant code in the `fill` function

Target	Aori		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `fill` function handles the transfer of native output tokens and ERC-20 tokens separately.

```
function fill(Order calldata order)
    external payable nonReentrant whenNotPaused onlySolver {
    [...]
    // Handle native or ERC20 output
    if (order.outputToken.isNativeToken()) {
        // For native tokens, solver must send exact amount via msg.value
        require(msg.value == order.outputAmount, "Incorrect native amount
    sent");
        order.outputToken.safeTransfer(order.recipient, order.outputAmount);
    } else {
        // For ERC20 tokens, ensure no native tokens were sent
        require(msg.value == 0, "No native tokens should be sent for ERC20
    fills");
        IERC20(order.outputToken).safeTransferFrom(msg.sender,
    order.recipient, order.outputAmount);
    }
    [...]
}
```

However, the `safeTransfer` function from the NativeTokenUtils library, which is used for native token transfers, already supports transferring both native and ERC-20 tokens. As a result, the separate logic in the `fill` function is redundant, and the `safeTransfer` function could be used to handle this logic.

```
function safeTransfer(address token, address to, uint256 amount) internal {
    if (isNativeToken(token)) {
        (bool success, ) = payable(to).call{value:  amount}("");
        require(success, "Native transfer failed");
    } else {
        IERC20(token).safeTransfer(to, amount);
    }
}
```

Impact

This redundant code adds unnecessary complexity.

Recommendations

We recommend removing the duplicated logic in the `fill` function and using the `safeTransfer` function for handling both ERC-20 and native token transfers.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `ff76040c` ↗.

3.5. Redundant native token–handling logic in `_executeDstHook` function

Target	Aori		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The internal `_executeDstHook` function processes hook execution during the `fill` function call. The trusted solver provides the necessary data for the fund transferring before the hook call. This data includes the `preferredToken` address and the `preferedDstInputAmount` (amount of tokens to be transferred from the solver to the specified hook address).

The `_executeDstHook` function supports both native and ERC-20 tokens. As a result, it utilizes the provided `msg.value` as input tokens for hook execution. However, it handles the case of nonzero `msg.value` and `preferredToken.isNativeToken()` separately using if/if-else blocks. Since the `msg.value > 0` condition is checked first, the block for `preferredToken.isNativeToken()` cannot be executed at all, making it effectively unreachable.

```
function _executeDstHook(
    Order calldata order,
    IAori.DstHook calldata hook
) internal allowedHookAddress(hook.hookAddress) returns (uint256 balChg) {
    if (msg.value > 0) {
        // Native token input
        (bool success, ) = payable(hook.hookAddress).call{value:
    msg.value}("");
        require(success, "Native transfer to hook failed");
    } else if (hook.preferedDstInputAmount > 0) {
        // ERC20 or native token input
        if (hook.preferredToken.isNativeToken()) {
            require(msg.value == hook.preferedDstInputAmount, "Incorrect
    native amount for preferred token");
            (bool success, ) = payable(hook.hookAddress).call{value:
    hook.preferedDstInputAmount}("");
            require(success, "Native transfer to hook failed");
        } else {
            IERC20(hook.preferredToken).safeTransferFrom(
                msg.sender,
                hook.hookAddress,
                hook.preferedDstInputAmount
            );
            }
        }
        [...]
    }
```

Impact

The unreachable code creates unnecessary complexity.

Recommendations

We recommend simplifying the logic by relying only on the `hook.preferredToken.isNativeToken()` check, which implicitly covers the case where native tokens are sent. An updated implementation could look like this.

```
function _executeDstHook(
    Order calldata order,
    IAori.DstHook calldata hook
) internal allowedHookAddress(hook.hookAddress) returns (uint256 balChg) {
    if (msg.value > 0) {
        // Native token input
        (bool success, ) = payable(hook.hookAddress).call{value: msg.value}("
            ");
        require(success, "Native transfer to hook failed");
    } else if (hook.preferedDstInputAmount > 0) {
   if (hook.preferedDstInputAmount > 0) {
        // ERC20 or native token input
        if (hook.preferredToken.isNativeToken()) {
            require(msg.value == hook.preferedDstInputAmount, "Incorrect
    native amount for preferred token");
            (bool success, ) = payable(hook.hookAddress).call{value:
    hook.preferedDstInputAmount}("");
            require(success, "Native transfer to hook failed");
        } else {
           require(msg.value == 0, "Native tokens should not be provided");
            IERC20(hook.preferredToken).safeTransferFrom(
                msg.sender,
                hook.hookAddress,
                hook.preferedDstInputAmount
            );
        }
    }
```

```
        } else {
            require(msg.value == 0, "Native tokens should not be provided");
        }
        [...]
}
```

Remediation

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

- `ff76040c`
- `869b4d1c`

3.6. Outdated version in EIP-712 domain separator

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

Description

The Aori contract has been updated to version 0.3.1, but the version string returned by `_domainNameAndVersion` remains "0.3.0".

```
function _domainNameAndVersion()
    internal
    pure
    override
    returns (string memory name, string memory version)
{
    return ("Aori", "0.3.0");
}
```

Impact

Users signing messages may see incorrect version information in their wallet's signature request, potentially causing confusion about which contract version they are interacting with.

Recommendations

We recommend updating the version string in the EIP-712 domain separator to reflect the current contract version.

```
function _domainNameAndVersion()
    internal
    pure
    override
    returns (string memory name, string memory version)
{
    return ("Aori", "0.3.0");
    return ("Aori", "0.3.1");
}
```

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `8c931f44` ↗.

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. Checks-effects-interactions pattern

Several functions in the codebase update the order status after performing external calls:

- `deposit` (with and without hook)
- `fill` (with and without hook)
- `cancel` (from the destination chain)

Additionally, the `withdraw` function reduces the unlocked balance after transferring tokens.

While all functions except `cancel` are protected by a reentrancy guard (see Finding 3.1. ↗), we still recommend following the checks-effects-interactions ↗ pattern. Contract state changes, such as updating balances or order status, should be made before performing any external calls. Following this pattern increases resilience against reentrancy risks and helps ensure safety even if protections like reentrancy guards are later removed or missed.

This issue has been acknowledged by Aori, and a fix was implemented in commit `658d3cd5` ↗.

4.2. Unnecessary signature in `depositNative`

The `depositNative` function accepts a signature and verifies that it is signed by the `order.offerer`. However, since the offerer can call this function directly, the signature is unnecessary and can be removed.

This issue has been acknowledged by Aori, and a fix was implemented in commit `331541e8` ↗.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: Aori contract (version 0.3.1 upgrade)

Description

The Aori contract is an intent settlement protocol designed to facilitate token exchanges across different blockchains between users and trusted solvers. Users can deposit tokens on a source chain with signed intent parameters, which solvers (market makers) can fulfill on destination chains. The contract also supports hooks, which are external contract calls executed during the deposit or fill process. This feature enables more complex operations, such as swapping one token for another before a deposit or using a DEX to acquire the required output token during a fill.

Notable changes

The following outlines the notable changes in version 0.3.1:

1. **Native token support**
    - The accounting system now uses `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` as the representative address for native tokens.
    - The `depositNative` function was added, allowing users to deposit native tokens to their locked balance.
    - The NativeTokenUtils library was introduced in AoriUtils.sol to manage native token transfers and monitor balance changes.
2. **Cancellation changes**
    - Source-chain cancellations for cross-chain orders were removed.
    - The `_cancel` function was updated to transfer tokens directly to the user instead of increasing their unlocked balance.
    - Recipients can now cancel expired cross-chain orders from the destination chain.
3. **Emergency functions**
    - The `emergencyWithdraw` function was added, enabling the contract owner to withdraw tokens from a specific user's locked or unlocked balance to maintain accounting consistency during emergencies.
    - The `emergencyCancel` function was introduced, allowing the contract owner to cancel orders from the source chain in emergency situations.

4. **Withdraw-amount input specification**
    - Inputting `0` as the `amount` in the `withdraw` function allows the caller to withdraw their full balance.
5. **Removed `swap()` function**
    - The unused `swap()` function for single-chain atomic swaps was removed to reduce contract size.

6. Assessment Results

During our assessment on the scoped Aori 0.3.1 Upgrade contracts, we discovered six findings. No critical issues were found. Two findings were of medium impact and the remaining findings were informational in nature.

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.