Zellic

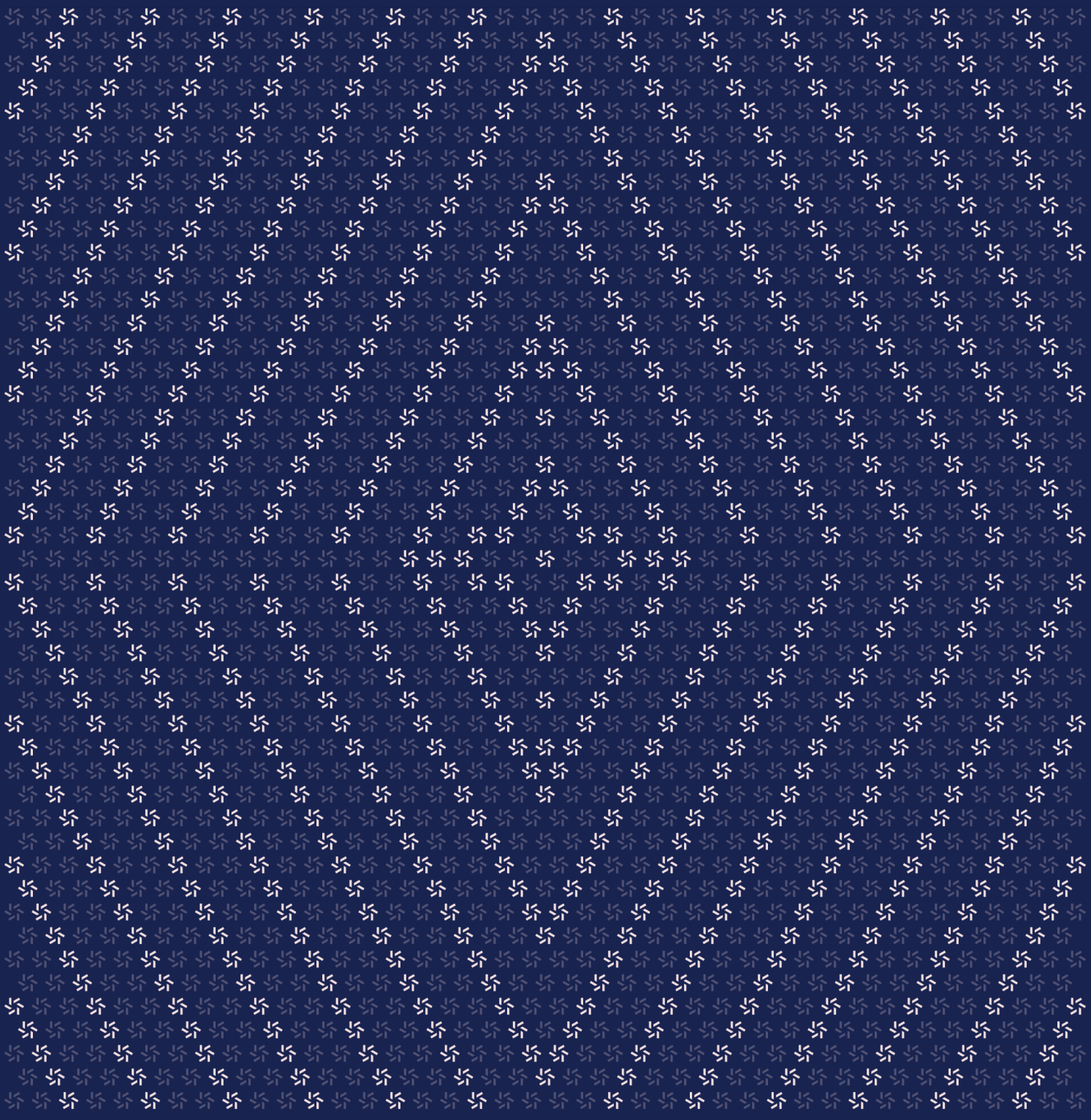
Prepared for
Ahmet Ozcan
Odos

Prepared by
Katerina Belotskaia
Sunwoo Hwang
Zellic

June 23, 2025

Odos Cross-Chain Contracts

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 Odos from June 17th to June 18th, 2025. During this engagement, Zellic reviewed Odos cross-chain contracts' 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:

- Does the protocol behave as expected?
- Is it possible for a malicious actor to exploit the bridging and swap logic to gain unauthorized access to funds?
- Could user funds become permanently locked?

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 Odos cross-chain contracts, we discovered two findings. No critical issues were found. One finding was of high impact and one was of medium impact.

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

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Odos Cross-Chain Contracts

Odos contributed the following description of Odos cross-chain contracts:

> This project provides a suite of smart contracts designed to facilitate seamless interaction between Odos and the Across Protocol. It enables cross-chain asset transfers and message passing by handling the necessary hook logic for initiating bridge transfers and handler logic for processing messages received from Across on a destination chain.

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:

Odos Cross-Chain Contracts

Type	Solidity
Platform	EVM-compatible
Target	odos-cross-chain-contracts
Repository	https://github.com/odos-xyz/odos-cross-chain-contracts ↗
Version	`399776e92a72f3882ce10346d1ac81438f4d4736`

Programs	`Across.sol`
	`AcrossBase.sol`
	`AcrossHandler.sol`
	`AcrossHook.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three 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 ↗

Sunwoo Hwang
Engineer
sunwoo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 17, 2025	Start of primary review period
June 18, 2025	End of primary review period

3. Detailed Findings

3.1. Inconsistent handling of native token

Target	AcrossHook.sol		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The OdosRouter and AcrossHook contracts use the zero address to represent the native token when interacting with the SpokePool contract. When the input token is the zero address, these contracts call the SpokePool's `deposit` function with value in Ether.

```
function _executeSpokePoolCall(address inputToken, uint256 amount,
    bytes memory callData) internal {
    if (inputToken != address(0)) {
        IERC20(inputToken).safeIncreaseAllowance(spokePool, amount);
    }

    (bool success, bytes memory returnData) = spokePool.call{value: inputToken
    == address(0) ? amount : 0}(callData);
// [...]
```

However, the SpokePool contract expects the WETH address for native token operations. The `deposit` function uses `msg.value` to convert Ether to WETH only when the input token matches the WETH address; otherwise, it requires `msg.value` to be 0. Consequently, transactions using the zero address as the input token will be reverted.

```
if (params.inputToken == address(wrappedNativeToken).toBytes32() && msg.value
    > 0) {
    if (msg.value != params.inputAmount)
    revert MsgValueDoesNotMatchInputAmount();
    wrappedNativeToken.deposit{ value: msg.value }();
    // Else, it is a normal ERC20. In this case pull the token from the caller
    as per normal.
    // Note: this includes the case where the L2 caller has WETH (already
    wrapped ETH) and wants to bridge them.
    // In this case the msg.value will be set to 0, indicating a "normal" ERC20
    bridging action.
} else {
    // msg.value should be 0 if input token isn't the wrapped native token.
    if (msg.value != 0) revert MsgValueDoesNotMatchInputAmount();
```

```
// [....]
```

Impact

All transactions using the zero address as the input token will fail due to the incompatible native-token handling between the AcrossHook and SpokePool contracts. This misalignment violates the core design requirement that token addresses should not be zero, as specified in the AcrossHandler's invariants.

Recommendations

Replace the zero address with the WETH address when interacting with the SpokePool contract to maintain consistency with the SpokePool's native-token–handling mechanism.

Remediation

This issue has been acknowledged by Odos, and a fix was implemented in PR #3 ↗.

3.2. Lack of validation of `outputToken` address

Target	AcrossHook.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In the `executeOdosHook` function of the AcrossHook contract, there is a lack of validation to ensure that the `outputToken` specified by the user in `hookData` is not the zero address. Since the zero address is used in OdosRouterV3 to represent native tokens, users may assume that it is a valid and supported value. However, on the destination chain, the `handleV3AcrossMessage` function, which processes the bridged message and tokens received from the SpokePool contract, will revert if `tokenSent` (equivalent to the `outputToken` from the sending side) is the zero address.

```
function handleV3AcrossMessage(address tokenSent, uint256 amount, address,
    bytes memory message)
    external
    virtual
    onlySpokePool
{
    require(tokenSent != address(0), "AcrossHandler: tokenSent cannot be the
    zero address");
    [...]
}
```

Impact

If the user specifies the zero address to indicate native-token usage, the swap process on the destination chain will fail, and the bridging and swap will not be completed.

Recommendations

We recommend adding a validation check in the `executeOdosHook` function to ensure that `outputToken` is not the zero address.

Remediation

This issue has been acknowledged by Odos, and a fix was implemented in PR #3.

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. Test suite

While the test suite covers the core contract logic, it lacks integration tests that validate correct interaction with the actual SpokePool contract.

This makes it harder to ensure that the contract behaves as expected, especially in scenarios that involve specific token handling or execution flows used by the real SpokePool.

For example, SpokePool expects the address of the wrapped native token to be provided when native tokens are used, then the OdosRouter and AcrossHook contracts use the zero address. As a result, native token handling logic behaves incorrectly in a real environment. See Finding 3.1. ↗ for more detail.

Consider extending the test suite to include integration tests with the actual SpokePool contract to ensure correct behavior when interacting with the real protocol implementation.

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. AcrossHook

The AcrossHook contract is responsible for handling hook functionality after swaps in the OdosRouter contract. It implements the `executeOdosHook` function, which can only be called by the OdosRouter contract. This function accepts `hookData` bytes, which are provided by the initiator of the swap action in the OdosRouter contract.

The `hookData`, which is fully controlled by the user, includes the following information:

```
struct AcrossDepositWithFee {
    address depositor;
    address recipient;
    address inputToken;
    address outputToken;
    uint256 inputAmount;
    uint256 outputAmount;
    uint256 destinationChainId;
    address exclusiveRelayer;
    uint32 quoteTimestamp;
    uint32 fillDeadline;
    uint32 exclusivityParameter;
    bytes message;
    uint256 feePercentage;
    address feeRecipient;
}
```

During `executeOdosHook` execution, it sends fees to the `feeRecipient` specified by the user and then calls the `deposit` function in the SpokePool contract using tokens received from the OdosRouter contract to initiate the cross-chain bridging of both the message and funds.

The `outputToken` address and `outputAmount` that the `recipient` will receive on the destination chain are also taken from `hookData`. While `executeOdosHook` does not validate these parameters, it does scale the `outputAmount` based on the actual input amount after fees and the `inputAmount` provided by the user. Additionally, the `executeOdosHook` function facilitates cross-chain swaps; the deposited amount can be used for a swap on the destination chain. However, this only occurs if the specified `recipient` on the destination chain is a smart contract, specifically, the AcrossHandler. If the `recipient` is an externally owned account, then the output tokens will simply be sent directly to the `recipient`, without any further logic execution.

Invariants

- The caller must be the OdosRouter contract.
- The fee percentage must be less than 10,000.
- The fee-recipient address cannot be the zero address.
- The token amount must be greater than zero.

5.2. AcrossHandler

The AcrossHandler contract is responsible for handling messages from the Across SpokePool. It implements the `handleV3AcrossMessage` function, which can only be called by the Across SpokePool contract. Before the function call, the SpokePool sends tokens, and then AcrossHandler calls the `swap` function in the OdosRouter contract using the provided tokens and message data.

Invariants

- The caller must be the Across SpokePool contract.
- The token address cannot be the zero address.
- The token amount must be greater than zero.

5.3. Important design aspects for users

The points outlined below are intended to highlight key aspects of the protocol's design that may affect user experience. These are not security vulnerabilities, but they are important for understanding how the protocol behaves in certain situations and where user awareness is critical for successful interaction.

Misconfigured `hookData` leads to unprocessed bridging or failed swap after bridging

The Across contract relies on the user to configure the `hookData` correctly. The `executeOdosHook` function on the sending side performs minimal validation, so the user can provide arbitrary or incorrect data — for example, an unsupported chain ID or an unreasonable `outputAmount`, and the function execution will not revert.

However, in particular, due to the SpokePool design, if unreasonable output amount is provided, relayers may never bridge the user's funds and message, leaving them unprocessed on the source chain. Or, if invalid output token or mismatched amounts of tokens were specified, the swap on the destination chain will fail. So it is the user's responsibility to ensure that the `hookData` is valid and correctly structured to avoid failed swaps or unprocessed bridging.

Available funds can be used by other users

Funds bridged to the destination chain but not claimed due to misconfigured swap data remain on the AcrossHandler contract. These funds are not locked or isolated and may be used by other users in subsequent swap executions. This behavior is part of the current design and should be clearly communicated to users.

Only wrapped native tokens are transferred to the `recipient` when it is a smart contract

When performing cross-chain swaps involving native tokens, the SpokePool contract on the destination chain only provides the wrapped version of the native token to the `recipient`, when it is a smart contract. As a result, the `recipient` is responsible for manually unwrapping the token if they need native assets after bridging.

Dummy data in the `deposit` function

When calling the `deposit` function in the SpokePool contract, the AcrossHook contract appends the dummy data `0x1dc0de0080` to the calldata. While this data does not affect the function's execution, it could confuse users. In response, Odos clarified that

> hex"1dc0de0080" is an `integratorId` that across requested us to append to the end of the calldata to track our txs.

```
        bytes memory callData = abi.encodeWithSelector(
            SPOKEPOOL_DEPOSIT_SELECTOR,
            params.depositor,
            params.recipient,
        // [...]
        );
@>        return bytes.concat(callData, hex"1dc0de0080");
```

6. Assessment Results

During our assessment on the scoped Odos cross-chain contracts, we discovered two findings. No critical issues were found. One finding was of high impact and one was of medium impact.

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.