Zellic

Odos

Smart Contract Security Assessment

June 5, 2023

Prepared for:

Yuri Papadin

Odos

Prepared by:

Filippo Cremonese and Ulrich Myhre

Zellic Inc.

Contents

About Zellic

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

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

1 Executive Summary

Zellic conducted a security assessment for Odos from May 24th to May 26th, 2023. During this engagement, Zellic reviewed Odos's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

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

- Are slippage checks secure against possible attacks?
- Can the contract be abused to access assets that do not belong to the caller?
- Are fees collected by the contract secure against an attacker?
- Is the referral mechanism safe against a malicious referrer?

1.2 Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody
- Incorrect implementation of other out-of-scope components of the system, such as the executors

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

The scope of this assessment was limited to the Router contract and did not include any other Odos component, both on and off chain. The analysis of the router was conducted with a limited understanding of its interactions with other components of the system.

1.3 Results

During our assessment on the scoped Odos Contracts, we discovered no findings.

Additionally, Zellic recorded its notes and observations from the assessment for Odos's benefit in the Discussion section (3) at the end of the document.

> During the remediation phase of the audit, Odos added an additional parameter to the `Swap` event in commit 77ef60b1. We conducted a review of this addition and did not identify any security vulnerabilities.

Breakdown of Finding Impacts

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

2 Introduction

2.1 About Odos

Odos Smart Order Routing (SOR) is a patented automated market maker (AMM) pathfinding algorithm, which aggregates decentralized exchanges (DEX) and finds optimal routes for cryptocurrency token swaps. Odos enables retail and institutional traders to benefit from incremental savings when converting one or several cryptocurrency tokens into other asset(s).

Zellic audited version two of the Odos Router contract.

2.2 Methodology

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

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

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

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

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

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

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

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

2.3 Scope

The engagement involved a review of the following targets:

Odos Contracts

Repository	https://github.com/odos-xyz/odos-router-v2
Version	odos-router-v2: `0e37bbd65a0ead242f7ecc4f562a617ec07e5bc8`
Program	• OdosRouterV2
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six engineer-days. The assessment was conducted over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Ulrich Myhre, Engineer
unblvr@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

May 26, 2023	Kick-off call
May 26, 2023	Start of primary review period
May 28, 2023	End of primary review period

3 Discussion

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

3.1 Calls to arbitrary external addresses

The contract performs calls to arbitrary external addresses for multiple reasons. While this is required in some contexts (e.g., working with any token contract without maintaining an allowlist) and we could not identify any exploitable vulnerability, we would suggest to allowlist addresses to which external calls are performed. In particular, it might be worthwhile to consider allowlisting the addresses of known executors, given their extremely sensitive role.

3.2 Fees on positive slippage

We note that the contract is written to keep any profit resulting from positive slippage instead of sharing the upside of a profitable trade with the user.

3.3 Hardcoded storage slot number

The `addressListStart` constant is set to the storage slot of the first element of the dynamic `addressList` array. This practice is error prone as the hardcoded value must be updated as the storage layout of the contract changes during development.

The following snippet demonstrates how it is possible to initialize an immutable variable with the storage slot of the first element of the array without hardcoding it.

```
pragma solidity ^0.8.13;

contract Storage {
    address[] addressList;
    uint256 private immutable addressListStart;

    constructor() {
        // Computed according to Solidity documentation
        // https://docs.soliditylang.org/en/v0.8.17/internals/layout_in
        // _storage.html#mappings-and-dynamic-arrays
        uint _addressListStart;
        assembly { _addressListStart := addressList.slot }
        addressListStart
    = uint256(keccak256(abi.encode(_addressListStart)));

        addressList.push(address(0x123));
        addressList.push(address(0x456));
        addressList.push(address(0x789));
    }

    function getSlot() public view returns (uint i) {
        // Accessing immutable variables directly from YUL is not
    supported
        uint _addressListStart = addressListStart;
        assembly {
            i := _addressListStart
        }
    }

    function read(uint idx) public view returns (address addr) {
        return addressList[idx];
    }

    function read_asm(uint idx) public view returns (address addr) {
        // Accessing immutable variables directly from YUL is not
    supported
        uint _addressListStart = addressListStart;
        assembly {
            addr := sload(add(_addressListStart, idx))
        }
    }
}
```

3.4 Coding style can lead to confusion

The code makes use of short if clauses without explicit scoping, which can lead to some confusion during development and auditing. One example is the following snippet from the function `_swapMulti()`.

```
referralInfo memory thisReferralInfo;
    if (referralCode > REFERRAL_WITH_FEE_THRESHOLD) thisReferralInfo
    = referralLookup[referralCode];
    {
      uint256 valueOut;
      uint256 _swapMultiFee = swapMultiFee;
      amountsOut = new uint256[](outputs.length);
       ...
    }
```

Due to the use of scoping inside the function, the one-liner if sentence, and no white space between those two, it is easy at first glance to assume that the scope is only executed if the clause is true, when in reality the code is equivalent to this.

```
referralInfo memory thisReferralInfo;
    if (referralCode > REFERRAL_WITH_FEE_THRESHOLD) {
      thisReferralInfo = referralLookup[referralCode];
    }

    // New scope
    {
      uint256 valueOut;
      uint256 _swapMultiFee = swapMultiFee;
      amountsOut = new uint256[](outputs.length);
       ...
    }
```

We recommend being explicit in the code where possible and especially in situations like the above where there is no gas cost or code size optimization driving the coding style.

> During the remediation phase of the audit, Odos added explicit scoping in commit e7f4c7af.

4 Threat Model

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

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

4.1 Module: OdosRouterV2.sol

Function: `registerReferralCode(uint32 _referralCode, uint64 _referralFee, address _beneficiary)`

This unpermissioned function allows to register a new referral code. Referral codes can have a referral fee, capped at max 2%.

Inputs

- `_referralCode`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be an existing referral code.
  - **Impact**: Integer identifying the referrer.
- `_referralFee`
  - **Control**: Arbitrary.
  - **Constraints**: Must represent a fee of at most 2%; must be zero if `_referralCode ≤ REFERRAL_WITH_FEE_THRESHOLD`.
  - **Impact**: Determines the referral fee.
- `_beneficiary`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be the null address.
  - **Impact**: Address of the referrer — receives the fees generated by the referral code.

Branches and code coverage (including function calls)

Intended branches

- Registers the new referral code.

☑ Test coverage

Negative behavior

- Reverts if the fee is > 2%.
  ☑ Negative test
- Reverts if the fee is > 0% and `_referralCode` ≤ `REFERRAL_WITH_FEE_THRESHOLD`.
  ☑ Negative test
- Reverts if the fee is 0% and `_referralCode > REFERRAL_WITH_FEE_THRESHOLD`.
  ☑ Negative test
- Reverts if the referral code is already registered.
  ☑ Negative test
- Reverts if the beneficiary address is the null address.
  ☑ Negative test

Function: `setSwapMultiFee(uint256 _swapMultiFee)`

This function allows the contract owner to specify the fee incurred by the users when performing multihop swaps.

Inputs

- `_swapMultiFee`
  - **Control**: Arbitrary.
  - **Constraints**: Must represent a fee of at most 0.5%.
  - **Impact**: Determines the new multihop fee.

Branches and code coverage (including function calls)

Intended branches

- Stores the new multihop fee.
  ☑ Test coverage

Negative behavior

- Reverts if the caller is not the contract owner.
  ☑ Negative test
- Reverts is the swap fee is too high.
  ☑ Negative test

Function: `swapCompact()`

This function allows the caller to perform a swap, encoding the details of the operation to be performed in a more efficient fashion than the plain `swap` function.

The function directly decodes inputs from `calldata`, ultimately building the same arguments supplied to `swap`. Space efficiency is achieved by packing the data as well as by allowing an efficient representation of zero addresses and use of addresses taken from a predefined list of addresses (which can be updated by the contract owner).

Branches and code coverage (including function calls)

Intended branches

- `getAddress` correctly handles compact zero addresses.
  - ☑ Test Coverage
- `getAddress` correctly handles addresses to be taken from the predefined list.
  - ☑ Test Coverage
- `getAddress` correctly handles full addresses.
  - ☑ Test Coverage
- Performs a swap using ETH as input.
  - ☑ Test Coverage
- Performs a swap using a token as input.
  - ☑ Test Coverage

Negative behaviour

- Execution is reverted if `inputAmount` does not match the message value.
  - ☑ Negative test
- Execution reverts if `inputToken == outputToken`.
  - ☑ Negative test
- Execution reverts if the minimum output amount is zero.
  - ☑ Negative test
- Execution reverts if the minimum output is less than the quote output.
  - ☑ Negative test
- Execution reverts if the output is too low (slippage).
  - ☑ Negative test

Function call analysis

- `rootFunction` → `_swapApproval`
  - **What is controllable?** All arguments.
  - **If return value controllable, how is it used and how can it go wrong?** Not

used.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards; reentrancy is a potential concern, but slippage checks mitigate that.

Function: `swapMultiPermit2(permit2Info permit2, inputTokenInfo[] inputs, outputTokenInfo[] outputs, uint256 valueOutMin, byte[] pathDefinition, address executor, uint32 referralCode)`

The multiswap variant of `swapPermit2`. This function performs multiple swaps by utilizing `PermitBatchTransferFrom` and `permitTransferFrom` with a signature from the owner of the tokens, instead of using the regular `safeTransferFrom`.

Inputs

- `permit2`
    - **Control**: Arbitrary.
    - **Constraints**: `signature` must be a valid signature for assets owned by `msg.sender` for the given nonce, deadline, and `transferDetails`.
    - **Impact**: Specifies the deadline, nonce, and signature used to validate the rest of the parameters.
- `inputs`
    - **Control**: Arbitrary.
    - **Constraints**: The sum of all `inputs[i].amountIn` must be equal to `msg.value` in the case of ETH. Cannot have duplicate sources.
    - **Impact**: Decides token addresses to swap from, the amounts to swap, and the receiver.
- `outputs`
    - **Control**: Arbitrary.
    - **Constraints**: Cannot be equal to any of the input token addresses (arbitrage) or have duplicates of destinations.
    - **Impact**: Specifies the output tokens, receiver, and `relativeValue` — which is a weighting for slippage calculations.
- `valueOutMin`
    - **Control**: Arbitrary.
    - **Constraints**: Must be larger than 0.
    - **Impact**: Decides the minimum value the token owner will allow with respect to slippage.
- `pathDefinition`
    - **Control**: Arbitrary.

  - **Constraints**: None.
  - **Impact**: Passed to the executor — determines the operation to be executed.
- `executor`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the address of the executor to be invoked.
- `referralCode`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Referral code used for statistics and referral fees.

Branches and code coverage (including function calls)

Intended branches

- Swapping ETH to tokens.
  - [x] Test coverage
- Swapping tokens to ETH.
  - [ ] Test coverage
- Swapping multiple tokens.
  - [ ] Test coverage

Negative behavior

- Execution is reverted if `msg.value == expected_msg_value`.
  - [ ] Negative test
- Same negative coverage as `swapMulti()`.
  - [ ] Negative test

Function call analysis

- `swapMultiPermit2 → _swapMulti`
  - **What is controllable?** All arguments.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are not caught and will make the entire swap revert.
- `swapMultiPermit2 → permitTransferFrom`
  - **What is controllable?** All arguments, except owner, which is always `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.

- **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are not caught and will make the entire swap revert.

Function: `swapMulti(inputTokenInfo[] inputs, outputTokenInfo[] outputs, uint256 valueOutMin, byte[] pathDefinition, address executor, uint32 referralCode)`

Swaps multiple tokens or ETH in a single atomically using approval. Slippage is controlled by giving each output a weight and defining a `valueOutMin` that defines a lower limit for the weighted sum of all output tokens.

Inputs

- `inputs`
  - **Control**: Arbitrary.
  - **Constraints**: The sum of all `inputs[i].amountIn` must be equal to `msg.value` in the case of ETH. Cannot have duplicate sources.
  - **Impact**: Decides token addresses to swap from, the amounts to swap, and the receiver.
- `outputs`
  - **Control**: Arbitrary.
  - **Constraints**: Cannot be equal to any of the input token addresses (arbitrage) or have duplicates of destinations.
  - **Impact**: Specifies the output tokens, receiver, and `relativeValue` — which is a weighting for slippage calculations.
- `valueOutMin`
  - **Control**: Arbitrary.
  - **Constraints**: Must be larger than 0.
  - **Impact**: Decides the minimum value the token owner will allow with respect to slippage.
- `pathDefinition`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Passed to the executor — determines the operation to be executed.
- `executor`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the address of the executor to be invoked.
- `referralCode`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: Referral code used for statistics and referral fees.

Branches and code coverage (including function calls)

Intended branches

- Performs a swap using ETH as input.
  - ☑ Test coverage
- Performs a swap using a token as input.
  - ☑ Test coverage
- Performs a swap using multiple tokens/ETH as input.
  - ☐ Test coverage

Negative behavior

- Execution is reverted if `inputs` has ETH but amount does not match `msg.value`.
  - ☑ Test coverage
- Execution is reverted if there are duplicate source tokens.
  - ☑ Test coverage
- Execution is reverted if there are any input and output tokens that are equal (arbitrage).
  - ☑ Test coverage
- Execution is reverted if slippage limit `valueOutMin` is exceeded (slippage).
  - ☑ Test coverage
- Execution is reverted if slippage limit is set too low.
  - ☑ Test coverage

Function call analysis

- `rootFunction` → `_swapMultiApproval`
  - **What is controllable?** All arguments.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards; reentrancy is a potential concern, but slippage checks mitigate that.

Function: `swapPermit2(permit2Info permit2, swapTokenInfo tokenInfo, byte[] pathDefinition, address executor, uint32 referralCode)`

This function allows to perform a swap, taking the assets to be used as input for the swap by using `permitTransferFrom` with a signature from the owner of the tokens instead of the regular `transferFrom`.

Inputs

- `permit2`
  - **Control**: Arbitrary.
  - **Constraints**: `signature` must be a valid signature for assets owned by `msg.sender` for the supplied nonce, deadline, token address, and amount.
  - **Impact**: Specifies part of the data required to call `permitTransferFrom` — the contract to call, nonce, deadline, and signature.
- `tokenInfo`
  - **Control**: Arbitrary.
  - **Constraints**: Same constraints as `swap`; additionally, the input token must be the same used by the signature.
  - **Impact**: Specifies the input and output tokens, their amounts, slippage, and receiver addresses.
- `pathDefinition`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Passed to the executor — determines the action to be executed.
- `executor`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the address of the executor contract.
- `referralCode`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Referral code used for statistics and referral fees.

Branches and code coverage (including function calls)

While only a basic test for this specific function is included in the test suite, we note the function shares the same logic used by `swap` and `swapCompact`, invoking the same `_swap` function. Therefore, only additional logic is documented here.

- ☑ Transfers input assets using `permitTransferFrom`, then invokes `_swap` to perform the swap.

Negative behavior

No negative tests exist in the codebase for this specific function; however, we note that most failure cases are shared with `swap` and `swapCompact`.

Function call analysis

- `rootFunction` → `permitTransferFrom)`
  - **What is controllable?** All arguments are controllable except the owner of the tokens, which is always `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards; reentrancy is a concern, mostly mitigated by slippage checks.

Function: `swapRouterFunds(inputTokenInfo[] inputs, outputTokenInfo[] outputs, uint256 valueOutMin, byte[] pathDefinition, address executor)`

This function allows the contract owner to swap the funds owned by the router (originating from fees and positive slippage).

Inputs

- `inputs`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Specifies the address and amount of the tokens to be used as input as well as the recipient of the transfer.
- `outputs`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Specifies the address, relative value, and recipient of the output tokens.
- `valueOutMin`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Specifies the value compared against the sum of relative values of the outputs for slippage protection.
- `pathDefinition`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Passed to the executor, specifies the operation to be performed.
- `executor`
  - **Control**: Arbitrary.
  - **Constraints**: None.

  - **Impact**: Specifies the address of the executor.

Branches and code coverage (including function calls)

We note that only a few basic tests for this function are performed.

Intended branches

- Transfers the inputs, invokes the executor, performs slippage checks, and transfers the outputs.
  - ☑ Test coverage

Negative behavior

- Reverts if the slippage is too high.
  - ☑ Negative test
- Reverts if the caller is not the owner.
  - ☑ Negative test

Function call analysis

- `rootFunction → _universalBalance(tokensIn[i])`
  - **What is controllable?** The address argument.
  - **If return value controllable, how is it used and how can it go wrong?** Controllable by specifying an arbitrary token, but this control has no meaningful use.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards. Reentrancy is possible but not exploitable.
- `rootFunction → _universalTransfer(tokensIn[i], ... )`
  - **What is controllable?** All arguments.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards. Reentrancy is possible but not exploitable.
- `rootFunction → balancesBefore[i] = _universalBalance(tokensOut[i])`
  - **What is controllable?** The address argument.
  - **If return value controllable, how is it used and how can it go wrong?** Controllable by specifying an arbitrary token, but this control has no meaningful use.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards. Reentrancy is possible but not exploitable.
- `rootFunction → IOdosExecutor(executor).executePath`
  - **What is controllable?** All arguments.

  - **If return value controllable, how is it used and how can it go wrong?**: Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts are propagated upwards. Reentrancy is possible but not exploitable.
- `rootFunction → amountsOut[i] = _universalBalance(tokensOut[i])`
  - **What is controllable?** The address argument.
  - **If return value controllable, how is it used and how can it go wrong?** Controllable by specifying an arbitrary token, but this control has no meaningful use.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards. Reentrancy is possible but not exploitable.
- `rootFunction → _universalTransfer(outputs[i].tokenAddress, ... )`
  - **What is controllable?** All arguments.
  - **If return value controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards. Reentrancy is possible but not exploitable.

Function: `swap(swapTokenInfo tokenInfo, byte[] pathDefinition, address executor, uint32 referralCode)`

This function allows the caller to perform a swap. The exact operation to be performed is encoded in the `pathDefinition`, and execution is delegated to the `executor` address. A referral code can be specified for statistical purposes as well as for collecting fee referral codes that support that.

Inputs

- `tokenInfo`
  - **Control**: Arbitrary.
  - **Constraints**: `inputAmount` field must be consistent with the value of the transaction.
  - **Impact**: Specifies the token (and amount) to be exchanged as well as the destination of the transfers.
- `pathDefinition`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Passed to the executor — determines the operation to be executed.
- `executor`
  - **Control**: Arbitrary.
  - **Constraints**: None.

    - **Impact**: Determines the address of the executor to be invoked.
- `referralCode`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Referral code used for statistics and referral fees.

Branches and code coverage (including function calls)

Intended branches

- Performs a swap using ETH as input.
  ☑ Test coverage
- Performs a swap using a token as input.
  ☑ Test coverage

Negative behavior

- Execution is reverted if `inputAmount` does not match the message value.
  ☑ Negative test
- Execution reverts if `inputToken == outputToken`.
  ☑ Negative test
- Execution reverts if the minimum output amount is zero.
  ☑ Negative test
- Execution reverts if the minimum output is less than the quote output.
  ☑ Negative test
- Execution reverts if the output is too low (slippage).
  ☑ Negative test

Function call analysis

- `rootFunction` → `_swapApproval`
    - **What is controllable?** All arguments.
    - **If return value controllable, how is it used and how can it go wrong?** Not used.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards; reentrancy is a potential concern, but slippage checks mitigate that.

Function: `transferRouterFunds(address[] tokens, uint256[] amounts, address dest)`

This function allows the contract owner to transfer assets owned by the router (originating from fees and positive slippage).

Inputs

- `tokens`
  - **Control**: Arbitrary.
  - **Constraints**: `tokens.length == amounts.length`.
  - **Impact**: List of token addresses to be transferred.
- `amounts`
  - **Control**: Arbitrary.
  - **Constraints**: `tokens.length == amounts.length`.
  - **Impact**: Amounts of the tokens to be transferred.
- `dest`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Receiver of the transferred tokens.

Branches and code coverage (including function calls)

Intended branches

- Transfers a specific amount of tokens.
  - [ ] Test coverage
- Transfers the full balance of tokens.
  - [ ] Test coverage
- Transfers a specific amount of ETH.
  - [x] Test coverage
- Transfers the full balance of ETH.
  - [x] Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - [x] Negative test

Function call analysis

- `rootFunction → _universalBalance`
  - **What is controllable?** The argument is controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is controllable by specifying an arbitrary token address; this has no meaningful consequence.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards. Reentrancy into the contract is possible

    but not a concern.

- `rootFunction → _universalTransfer`
  - **What is controllable?** All arguments are controllable.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are propagated upwards. Reentrancy into the contract is possible but not a concern.

Function: `writeAddressList(address[] addresses)`

This function allows the contract owner to append addresses to the address list used by the compact versions of the functions performing swaps.

Inputs

- `addresses`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array with the additional addresses to be appended to the address list.

Branches and code coverage (including function calls)

Intended branches

- Appends the new addresses to the address list.
  ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  ☑ Negative test

5 Audit Results

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

During our assessment on the scoped Odos Contracts, we discovered no findings.

5.1 Disclaimer

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

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

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