Zellic

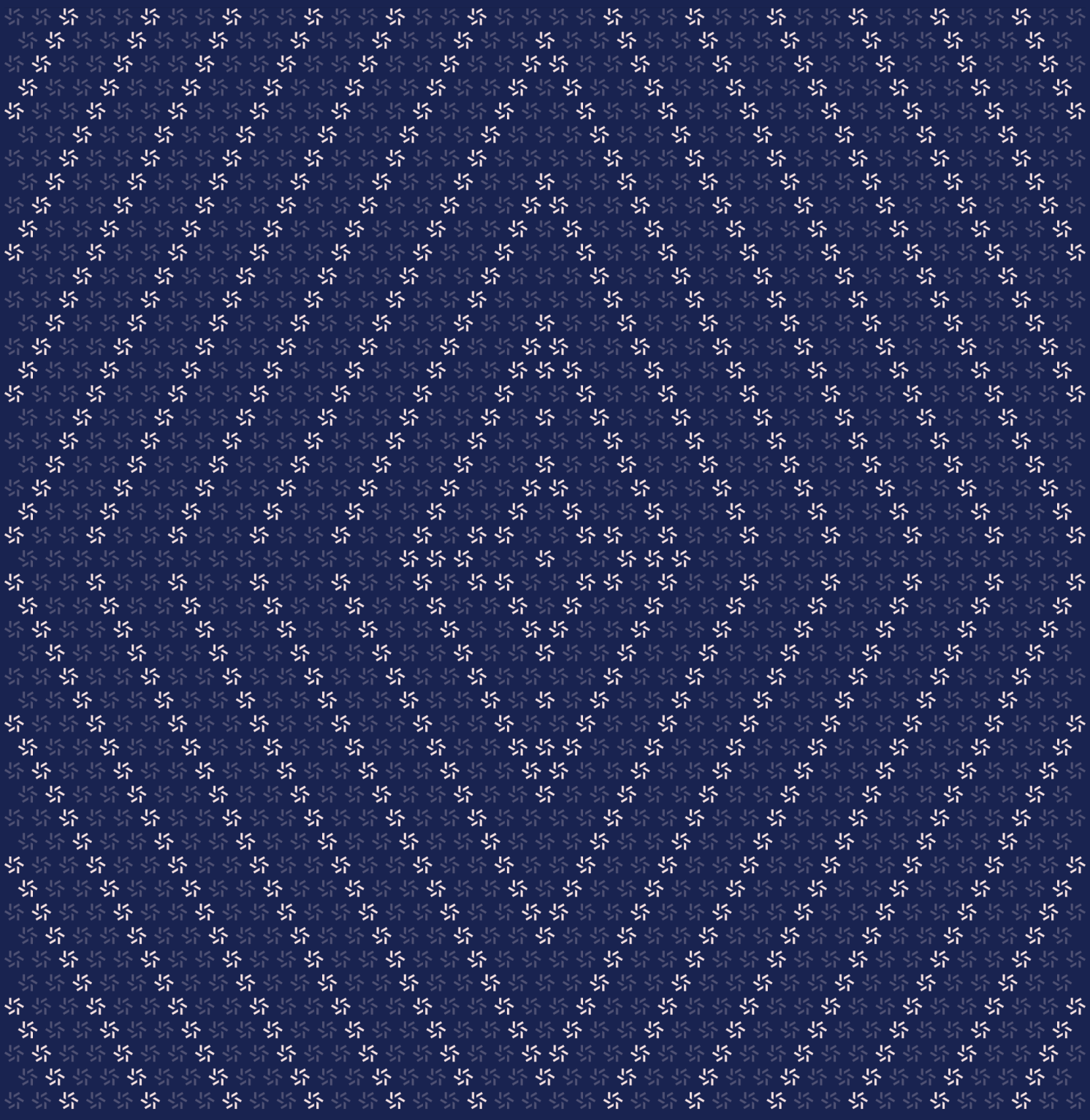
Prepared for
Hilliam Tung
Aori

Prepared by
Katerina Belotskaia
Dimitri Kamenski
Zellic

October 10, 2024

AoriV2

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 October 4th to October 7th, 2024. During this engagement, Zellic reviewed AoriV2'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:

- Assuming a user has enough funds, and approval and order is placed within a timely manner, can settlement contradict expected behavior?
- How can we secure flash loans against reentrancy or misuse?
- Are there attack vectors against the Uniswap-like V4 hooks that the protocol has not considered?

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

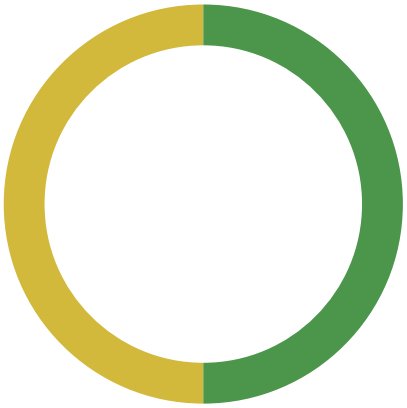
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped AoriV2 contracts, we discovered four findings. No critical issues were found. Two findings were of medium impact and two were of low impact.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	2
Low	2
Informational	0



2. Introduction

2.1. About AoriV2

Aori contributed the following description of AoriV2:

> Aori is a high-performance protocol facilitating high-frequency trading on-chain and OTC settlement. It focuses on allowing for any token to be traded through it, through the use of solvers, market making vaults or other actors fulfil orders as a maker and be the counterparty to trades. They do not need the assets at the time of matching but do for the time of settlement, which the protocol allows for them to fetch the required assets just-in-time to fulfill the user's trade.

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.

2.3. Scope

The engagement involved a review of the following targets:

AoriV2 Contracts

Type	Solidity
Platform	EVM-compatible
Target	aori-v2-contracts
Repository	https://github.com/aori-io/aori-v2-contracts ↗
Version	`9fbe63cdef00eff7853ea47b11c68fcd62659de8`
Programs	`AoriV2`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.5 person-days. The assessment was conducted by two consultants over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Dimitri Kamenski
Engineer
dimitri@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 4, 2024	Kick-off call
October 4, 2024	Start of primary review period
October 7, 2024	End of primary review period

3. Detailed Findings

3.1. Lack of balance verification in the `flashLoan` function

Target	AoriV2.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The AoriV2 contract provides users with the opportunity to use flash-loan functionality. The `flashLoan` function supports two modes, depending on the value of the `receiveToken` input parameter. If `receiveToken` is false, the flash loan will be applied to the internal recipient's `balances`. Otherwise, the requested tokens will be transferred directly to the `recipient` address. After that, the recipient's `receiveFlashLoan` function call will be executed. Finally, the funds will either be transferred back to the contract balance if `receiveToken` is true, or the internal recipient's `balances` will be increased by the requested `amount`. If there are not enough tokens in the `balances`, the transaction will be reverted. However, in the case of a direct token transfer, a balance check is not implemented.

```
function flashLoan(
    address recipient,
    address token,
    uint256 amount,
    bytes memory userData,
    bool receiveToken
) external {
    // Flash loan
    if (receiveToken) {
        IERC20(token).safeTransfer(recipient, amount);
    } else {
        balances[recipient][token] += amount;
    }

    // call the recipient's receiveFlashLoan
    IFlashLoanReceiver(recipient).receiveFlashLoan(
        token,
        amount,
        userData,
        receiveToken
    );

    if (receiveToken) {
        IERC20(token).safeTransferFrom(recipient, address(this), amount);
```

```
    } else {
        balances[recipient][token] -= amount;
    }
}
```

Impact

The AoriV2 contract implies the use of various implementations of ERC-20 tokens, including non-standard ones, where a successful `safeTransferFrom` function does not guarantee that the funds will be fully paid back.

Recommendations

Consider adding a check to ensure that the `token` balance after repayment is not less than that of before the flash-loan action.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit 367515e7 ↗.

3.2. The `serverSigner` address can be initialized to a zero address

Target	AoriV2.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The following assertion in the `settleOrders` function can be bypassed if the `serverSigner` address is zero, since `ecrecover` returns a zero address in the case of an error.

```
require(
    serverSigner ==
        ecrecover(
            keccak256(
                abi.encodePacked(
                    "\x19Ethereum Signed Message:\n32",
                    getMatchingHash(matching)
                )
            ),
            serverV,
            serverR,
            serverS
        ),
    "Server signature does not correspond to order details"
);
```

Impact

In the case that `serverSigner == address(0)` any invalid signature can be provided to bypass this verification.

Recommendations

This issue could be fixed by adding a zero-address check on the `_serverSigner` parameter.

```
constructor(address _serverSigner) {
    require(_serverSigner != 0, "_serverSigner can not be zero address");
    serverSigner = _serverSigner;
```

```
}
```

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit 77cf70c2 ↗.

3.3. Server signature can be reused more than once

Target	AoriV2.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Settling orders on AoriV2 assumes that the off-chain matching has been concluded, which requires a taker, maker, and server signature. The server signature is then passed through `ecrecover()` as an integrity check to ensure a valid server signer was used for the matching. Due to errors in the usage of `ecrecover`, it is possible to have equivalent signatures for different `r`, `s`, and `v` values of the server. However, protection against reuse of equivalent signatures prevents this issue from being exploited to double-spend user trades.

An ECDSA signature is represented by two 32-byte `r` and `s` values along with a `v` recovery value. R is derived from the ephemeral public key used during signing. S is calculated using the message hash, signer private key, and ephemeral key. As the curves themselves are symmetrical, there exists a matching signature for some `r'`, `s'`, and `v'`. As the values of `r`, `s`, and `v` are known and submitted on chain, it is possible that an attacker is able to modify the S and V values independently of each other, in order to produce the same signature under different inputs.

In the `settleOrders()` function, S values are unbounded whilst V values are incorrectly between 27 and 53, allowing the caller to change values of S and V with all other inputs constant. If the signing library expects S to be contained only in lower-bound order, an attacker can use an upper-bound S value. V values should be restricted to 27 or 28. (In some cases, a nonstandard library may use 0 and 1; therefore, we would scale it to 27 and 28.) However, the server-signing library would be known ahead of time and unlikely to change, so appropriate bounds should be placed that match expectations.

Impact

Signature malleability allows reuse of the same valid signature under differing R, S, and V values. However, the same signature would only be possible under the same `matching`, which would imply that the `BitMaps` for `orderStatus` has already been set for both the `taker` and `maker`. This prevents practical reuse of signatures in the `settleOrder()` function.

Recommendations

We recommend determining what S bounds are used by the off-chain server-signing library. Values of S should normally be bound to the lower half order as follows:

```
if (uint256(s) >
    0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
    return (address(0), RecoverError.InvalidSignatureS);
}
```

Additional checks should be placed to ensure that V is either 27 or 28.

Alternatively, consider using a vetted library such as ESCDA ↗ from OpenZeppelin.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit 8426fc07 ↗.

The team has also commented on the issue:

> The `ecrecover` has been replaced with OpenZeppelin's `SignatureChecker.isValidSignatureNow` that directly uses `ECDSA.tryRecover` to handle signature malleability attacks. This has the additional ability to use EIP-1271 signatures from a smart contract wallet e.g Safe Wallet as the `serverSigner` which is now out-of-protocol to deal with.

3.4. Nonstandard ERC-20 tokens may break trade accounting

Target	AoriV2.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The AoriV2 contract acts as an on-chain order book for P2P trading of ERC-20 tokens through an intermediary matching system. Documentation notes that the protocol is intended to support "any token". However, several counterexamples exist that compromise protocol TVL and user funds.

The protocol allows users to deposit and withdraw as follows:

```
function deposit(
    address _account,
    address _token,
    uint256 _amount
) external {
    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
    balances[_account][_token] += _amount;
}

function withdraw(address _token, uint256 _amount) external {
    balances[msg.sender][_token] -= (_amount);
    IERC20(_token).safeTransfer(msg.sender, _amount);
}
```

This functionality assumes that the user input for `_amount` and the real asset value deposited (or withdrawn) are equivalent at all times. There are many cases where nonstandard ERC-20 logic breaks this assumption; below are two examples.

1. cUSDCv3

First, cUSDv3 has unique transfer functionality that caps `type(uint256).max` at the maximum balance of the user initiating the transfer. As such, an attack can be created where a user deposits `type(uint256).max` into the protocol, which only transfers their balance (assume some dust amount). Therefore, their balance would be credited with `type(uint256).max`. This will allow them to withdraw their entire balance (executed as a transfer from the AoriV2 contract). This would withdraw the entire balance of the AoriV2 contract.

2. Fee-on-transfer tokens (rebasing)

Tokens that charge a fee on transfer will have differing values between what is transferred to the smart contract and the value initially input into the `deposit()` function.

Both these examples lead to inaccuracies in accounting as a result of the token input versus the token output of transfers not being monitored.

Impact

Nonstandard ERC-20 tokens may result in accounting inaccuracies that could lead to loss of user funds and missing protocol TVL.

Recommendations

There is a wide variety of nonstandard ERC-20 tokens with different edge cases. We recommend only incrementing or deducting user balances by the delta value after the transfer is recorded.

Additionally, it is worth considering what tokens the protocol wishes to support, opting to only support standard ERC-20–compliant tokens.

Remediation

This issue has been acknowledged by Aori, and a fix was implemented in commit `aedf91c7` ↗.

The team has also commented on the issue:

> Reentrancy lock added specifically for the `withdraw` function to ensure accurate delta accounting. In regards to the consideration of tokens that can be supported, we acknowledge that direct trading with custom transfer logic may cause issues e.g `cUSDCv3`'s max-transfer. For these, we'll focus on ensuring that users first deposit into the contract and operate with that in-contract balance to alleviate some of these edge cases to bypass such logic when performing the swap.

4. 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.

4.1. Module: AoriV2.sol

Function: `deposit(address _account, address _token, uint256 _amount)`

This function allows to deposit tokens to the contract before settlement execution.

Inputs

- `_account`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The address of the account where the deposited funds will be recorded.
- `_token`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The token to be deposited.
- `_amount`
  - **Control**: Full control.
  - **Constraints**: If `msg.sender` does not own enough tokens, the transaction will be reverted.
  - **Impact**: The amount of tokens to be deposited.

Branches and code coverage

Intended branches

- the `_account` balance was updated properly
  - ☑ Test coverage

Negative behavior

- fails without approval
  - ☑ Negative test
- not enough funds
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransferFrom(IERC20(_token), msg.sender, address(this), _amount)`
  - **What is controllable?** `_token` and `_amount`.
  - **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?** This is standard token transfer using the SafeERC20 library. But in the case of non-standard ERC-20 token implementation, unexpected behavior may occur.

Function: `flashLoan(address recipient, address token, uint256 amount, bytes userData, bool receiveToken)`

This function allows caller to make a flash loan. All funds should be returned at the end of the function execution. No fee is charged.

Inputs

- `recipient`
  - **Control**: Full control.
  - **Constraints**: Should be contract.
  - **Impact**: The receiver of the flash loan.
- `token`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The loaned token.
- `amount`
  - **Control**: Full control.
  - **Constraints**: The contract must have sufficient funds.
  - **Impact**: Loan amount.
- `userData`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The data for the `receiveFlashLoan` external call.
- `receiveToken`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: If true, the direct token transfer will be executed; otherwise, only the internal token balance will be updated.

Branches and code coverage

Intended branches

- flash loan was executed properly when `receiveToken` is true
  - ☑ Test coverage
- flash loan was executed properly when `receiveToken` is false
  - ☑ Test coverage

Negative behavior

- recipient is not a contract or doesn't support `receiveFlashLoan`
  - ☑ Negative test
- the funds were not refunded in full
  - ☐ Negative test
- not enough liquidity
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(token), recipient, amount)`
  - **What is controllable?** `token`, `amount`, and `recipient`.
  - **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?** This is standard token transfer using the SafeERC20 library.
- `IFlashLoanReceiver(recipient).receiveFlashLoan(token, amount, userData, receiveToken)`
  - **What is controllable?** `token`, `amount`, `userData`, and `receiveToken`.
  - **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?** Reentrancy is possible, but there are no impacts here.
- `SafeERC20.safeTransferFrom(IERC20(token), recipient, address(this), amount)`
  - **What is controllable?** `token`, `amount`, and `recipient`.
  - **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?** This is standard token transfer using the SafeERC20 library.

Function: `settleOrders(MatchingDetails matching, bytes serverSignature, bytes hookData)`

This function is intended for verifying orders, signatures of the maker, taker, and server signer. If all provided data is correct, assets will be transferred to recipients specified by the maker and taker. Also, a fee will be charged if the funds in the orders exceed the requested ones; this difference will be sent to the `feeRecipient` specified by the `serverSigner` side. It is assumed that the call is made by the maker because only the maker owns the `makerSignature` data. But in fact, the function does not validate the `msg.sender` address.

Inputs

- `matching`
  - **Control**: Full control.
  - **Constraints**: The maker and taker orders should match.
  - **Impact**: Contains maker and taker orders as well as `makerSignature`, `takerSignature`, and the address of the fee recipient.
- `serverSignature`
  - **Control**: Full control.
  - **Constraints**: There is a check that this is a valid `serverSigner` signature.
  - **Impact**: The `matching` data should be signed by the `serverSignature`.
- `hookData`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Arbitrary data for the `beforeAoriTrade` and `afterAoriTrade` hook calls of the maker contract.

Branches and code coverage

Intended branches

- the settlement was made properly
  - ☑ Test coverage
- the fee was charged properly
  - ☑ Test coverage
- the fee is not charged when there are no excess funds in the orders.
  - ☑ Test coverage

Negative behavior

- maker order was already used
  - ☑ Negative test
- taker order was already used
  - ☑ Negative test

- invalid maker, taker and server signer signatures
  - ☑ Negative test
- maker order `startTime` is in the future
  - ☑ Negative test
- taker order `startTime` is in the future
  - ☑ Negative test
- maker order `endTime` is in the past
  - ☑ Negative test
- taker order `endTime` is in the past
  - ☑ Negative test
- taker's and maker's `chainId` don't match
  - ☑ Negative test
- taker's and maker's `zone` don't match and are invalid
  - ☑ Negative test
- taker's and maker's `inputAmount/outputAmount` don't match
  - ☑ Negative test

Function call analysis

- `SignatureChecker.isValidSignatureNow(matching.makerOrder.offerer, {'keccak256'}(abi.encodePacked("\x19Ethereum Signed Message:\n32", makerHash)), abi.encodePacked(makerR, makerS, makerV))`
  - **What is controllable?** `matching.makerOrder.offerer`.
  - **If the return value is controllable, how is it used and how can it go wrong?** This function returns a boolean result for signature verification. In the case of an error during ECDSA signature recovery, the signature will be validated using ERC-1271. Only if both verifications fail, or an error occurs, will `isValidSignatureNow` return false. If one of the checks passes, the function will return true.
  - **What happens if it reverts, reenters or does other unusual control flow?** `staticcall` cannot be used for a reentrancy attack.
- `SignatureChecker.isValidSignatureNow(matching.takerOrder.offerer, {'keccak256'}(abi.encodePacked("\x19Ethereum Signed Message:\n32", takerHash)), abi.encodePacked(takerR, takerS, takerV))`
  - **What is controllable?** `matching.takerOrder.offerer`.
  - **If the return value is controllable, how is it used and how can it go wrong?** This function returns a boolean result for signature verification. In the case of an error during ECDSA signature recovery, the signature will be validated using ERC-1271. Only if both verifications fail, or an error occurs, will `isValidSignatureNow` return false. If one of the checks passes, the function will return true.
  - **What happens if it reverts, reenters or does other unusual control flow?** `staticcall` cannot be used for a reentrancy attack.
- `BitMaps.get(this.orderStatus, uint256(makerHash))`

  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns whether the given `makerHash` has already been written to the `orderStatus`.
  - **What happens if it reverts, reenters or does other unusual control flow?** No problems.
- `BitMaps.get(this.orderStatus, uint256(takerHash))`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns whether the given `takerHash` has already been written to the `orderStatus`.
  - **What happens if it reverts, reenters or does other unusual control flow?** No problems.
- `BitMaps.set(this.orderStatus, uint256(makerHash))`
  - **What is controllable?** N/A.
  - **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?** No problems.
- `BitMaps.set(this.orderStatus, uint256(takerHash))`
  - **What is controllable?** N/A.
  - **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?** No problems.
- `SafeERC20.safeTransferFrom(IERC20(matching.takerOrder.inputToken), matching.takerOrder.offerer, address(this), matching.takerOrder.inputAmount)`
  - **What is controllable?** N/A.
  - **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?** This is standard token transfer from a specified sender using the SafeERC20 library. Transfers taker's input amount to the contract.
- `SafeERC20.safeTransfer(IERC20(matching.makerOrder.outputToken), matching.makerOrder.recipient, matching.makerOrder.outputAmount)`
  - **What is controllable?** N/A.
  - **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?** This is standard token transfer using the SafeERC20 library. Transfers maker's output amount to the maker's recipient.
- `IERC165(matching.makerOrder.offerer).supportsInterface(IAoriHook.beforeAoriTrade.selector)`
  - **What is controllable?** N/A.

- **If the return value is controllable, how is it used and how can it go wrong?** The `supportsInterface` should return true in case the `beforeAoriTrade` function is supported by the `offerer` contract.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `IAoriHook(matching.makerOrder.offerer).beforeAoriTrade(matching, hookData)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns boolean result.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is possible, but it cannot be used to reuse the same signatures because they have already been marked in bitmap as used.
- `SafeERC20.safeTransferFrom(IERC20(matching.makerOrder.inputToken), matching.makerOrder.offerer, address(this), matching.makerOrder.inputAmount)`
  - **What is controllable?** N/A.
  - **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?** This is standard token transfer from a specified sender using the SafeERC20 library. Transfers maker's input amount to the contract.
- `SafeERC20.safeTransfer(IERC20(matching.takerOrder.outputToken), matching.takerOrder.recipient, matching.takerOrder.outputAmount)`
  - **What is controllable?** N/A.
  - **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?** This is standard token transfer using the SafeERC20 library. Transfers taker's output amount to the taker's recipient.
- `IERC165(matching.makerOrder.offerer).supportsInterface(IAoriHook.afterAoriTrade.selector)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** The `supportsInterface` should return true in case the `afterAoriTrade` function is supported by the `offerer` contract.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IAoriHook(matching.makerOrder.offerer).afterAoriTrade(matching, hookData)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns boolean result.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is possible, but it cannot be used to reuse the same signatures because they have already been marked in bitmap as used and this call occurs when all necessary contract states are updated.

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

This function allows the caller to withdraw assets from the contract.

Inputs

- `_token`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The token to be withdrawn.
- `_amount`
  - **Control**: Full control.
  - **Constraints**: If `msg.sender` does not own enough tokens, the transaction will be reverted due to an underflow error.
  - **Impact**: The amount of tokens to be withdrawn.

Branches and code coverage

Intended branches

- The `msg.sender` balance was updated properly.
  - ☑ Test coverage

Negative behavior

- There are not enough tokens on `msg.sender` to withdraw.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(_token), msg.sender, _amount)`
  - **What is controllable?** `_token` and `_amount`.
  - **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?** This is standard token transfer using the SafeERC20 library. But in the case of non-standard ERC-20 token implementation, unexpected behavior may occur.

5. Assessment Results

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

During our assessment on the scoped AoriV2 contracts, we discovered four findings. No critical issues were found. Two findings were of medium impact and two were of low impact.

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 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.