Zellic

Bebop Aggregation

Smart Contract Security Assessment

Februrary 2, 2023

Prepared for:

Bebop

Prepared by:

Ayaz Mammadov and Vlad Toie

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 Bebop from January 18th to January 20th, 2023. During this engagement, Zellic reviewed Bebop Aggregation'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:

- Can any of the signatures be forged or replayed?
- How is the order validation logic performed?
- How does Bebop Aggregation handle the case where a user's signature is invalid?
- Could a malicious user attempt to duplicate and/or steal funds from takers that have crafted signatures?

1.2 Non-goals and Limitations

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

- Problems relating to the front-end components and infrastructure of the project
- Any issues stemming from code or infrastructure outside of the assessment scope

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, we could not further investigate issues that may arise from the way the signatures are crafted in the first place. Although we focused on the order validation logic, we did not assess the logic that is used to craft the signatures in the first place. Because of that, the signature integrity depends on how the front-end is implemented and how the user interacts with the front-end or other off-chain components that may facilitate the generation of the signatures.

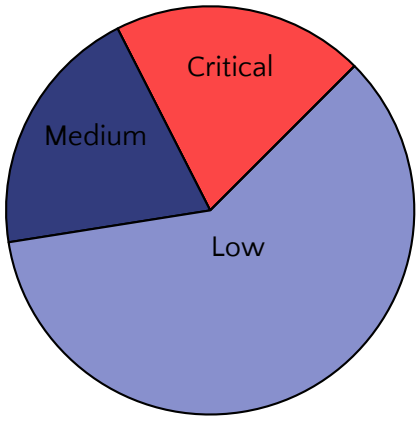
1.3 Results

During our assessment on the scoped Bebop Aggregation contracts, we discovered five findings. Of the five findings, one was of critical impact, one of medium impact, and the remaining findings were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Bebop's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Bebop Aggregation

Bebop Aggregation Smart Contracts is a multi-maker, multi-order, single-taker, trustless validation and settlement mechanism for all Bebop order types occurring with both split and nonsplit priced swaps.

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:

Bebop Aggregation Contracts

Repository	https://github.com/bebop-dex/bebop-smart-contracts/
Versions	`fa724361b7a2e1e623c4fca378536d2072fb1997`
Programs	• bebop_aggregation_contract.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

January 18, 2023	Kick-off call
January 18, 2023	Start of primary review period
January 20, 2023	End of primary review period
February 2, 2023	Closing call

3 Detailed Findings

3.1 Arbitrary trade forgery

- **Target**: bebop_aggregation_contract
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

A malicious user can forge an arbitrary trade including any trader.

```
function validateMakerSignature(
    address maker_address,
    bytes32 hash,
    Signature memory signature
) public view override{
    ...
    } else if (signature.signatureType == SignatureType.EIP1271) {
        require(IERC1271(signature.walletAddress).isValidSignature(hash,
    signature.signatureBytes) == EIP1271_MAGICVALUE, “Invalid Maker EIP
    1271 Signature”);
    ...
}
```

This is caused by the user-supplied maker signatures, which can be set to EIP1271 signatures, verified against a user-supplied wallet address that has to return a valid value. However, there is nothing that binds the maker addresses to the wallet addresses. As a result, a user can supply an arbitrary maker address and a wallet address that will always return the correct value to pass signature checks.

Impact

A malicious user can forge extremely unbalanced one-sided trades to steal funds from any user (market maker or taker) that has approval to the Bebop contract.

Recommendations

Bind the wallet address to the maker address or use the maker address as the wallet address.

Remediation

The issue has been fixed in commit ce63364f.

3.2 Any order can be blocked permanently

- **Target**: bebop_aggregation_contract
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

```
// Construct partial orders from aggregated orders
function assertAndInvalidateAggregateOrder(
    AggregateOrder memory order,
    bytes memory takerSig,
    Signature[] memory makerSigs
) public override returns (bytes32) {
```

The public function `assertAndInvalidateAggregateOrder` checks the validity of an order, but in doing so it also sets the nonce of that order. This function should be called by `SettleAggregateOrder` after which the trade is executed. If called alone, it will set the nonce of that specific trade, and as a result, that order cannot be used since the nonce will be set.

Impact

A user can block any person's call to `SettleAggregateOrder` by calling `assertAndInvalidateAggregateOrder` first.

Recommendations

Change the visibility of `assertAndInvalidateAggregateOrder` to `internal`.

Remediation

The issue has been fixed in commit fa724361.

3.3 A nonce of 0 can result in signature replay attacks

- **Target**: bebop_aggregation_contract
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The function `invalidateOrder` is responsible for checking and setting nonces in a gas-efficient manner; it does so by checking a certain slot, and if the slot is not 0, the nonce has been used.

```
function invalidateOrder(address maker, uint256 nonce) private {
    uint256 invalidatorSlot = uint64(nonce) >> 8;
    uint256 invalidatorBit = 1 << uint8(nonce);
    mapping(uint256 => uint256) storage invalidatorStorage
    = maker_validator[maker];
    uint256 invalidator = invalidatorStorage[invalidatorSlot];
    require(invalidator & invalidatorBit == 0, "Invalid maker order
    (nonce)");
    invalidatorStorage[invalidatorSlot] = invalidator | invalidatorBit;
}
```

However, the specific nonce 0 will always pass this check.

Impact

If the nonce 0 was chosen as the nonce to use, signature replay attacks could be possible, causing loss of funds for either a market maker or taker.

Recommendations

Enforce that the nonce supplied is never 0.

Remediation

The issue has been fixed in commit e4aa345b.

3.4 The signature may be too short

- **Target**: bebop_aggregation_contract
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

There are no checks on the signature length in the `getRsv` function. This function is responsible for extracting the r/s/v values from a signature. The function is defined as follows:

```
function getRsv(bytes memory sig) internal pure returns (bytes32,
    bytes32, uint8) {
    bytes32 r;
    bytes32 s;
    uint8 v;

    assembly {
        r := mload(add(sig, 32))
        s := mload(add(sig, 64))
        v := and(mload(add(sig, 65)), 255)
    }
    if (v < 27) v += 27;
    return (r, s, v);
}
```

In case the signature is shorter than 65 bytes, the r/s will be padded with zeroes, which could lead to undesired behavior.

Impact

The impact of this issue is low, as the function is only used to extract the r/s/v values from a signature. The function is not used to verify the signature itself, which is done by the `ecrecover` function.

Recommendations

We recommend adding a check that ensures the signature is 65 bytes long.

```
function getRsv(bytes memory sig) internal pure returns (bytes32,
    bytes32, uint8) {

    require(sig.length ≥ 65, “Signature too short”);

    bytes32 r;
    bytes32 s;
    uint8 v;

    assembly {
        r := mload(add(sig, 32))
        s := mload(add(sig, 64))
        v := and(mload(add(sig, 65)), 255)
    }
    if (v < 27) v += 27;
    return (r, s, v);
}
```

Remediation

The issue has been fixed in commit ba4a5804.

3.5 Signatures are malleable

- **Target**: bebop_aggregation_contract
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The signatures that are usable in the contract are theoretically malleable. This is because the `ecrecover` function does not check if the s value is in the lower half of the secp256k1 curve's order and does not check if the v value is either 27 or 28.

```
// get r, s and v from signature

    return ecrecover(hash, v, r, s);

    // ... no other checks are performed
```

Impact

Currently, there are no major security implications, since the nonce is used to prevent replay attacks. However, this could be a problem in the future, if the nonce is removed.

Recommendations

We recommend following the ECDSA's Library recommendation, and ensure that the s value is in the lower half of the secp256k1 curve's order as well as ensure that the v value is either 27 or 28.

```
// get r, s and v from signature

    if (uint256(s)
    > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0)
    {
        return (address(0), RecoverError.InvalidSignatureS);
    }
    if (v ≠ 27 && v ≠ 28) {
        return (address(0), RecoverError.InvalidSignatureV);
    }

    // ...
```

Remediation

The issue has been fixed in commit ba4a5804 by implementing checks in the functions `validateMakerSignature`, `assertAndInvalidateMakerOrders` and `assertAndInvalidateAggregateOrder`.

Update February 2, 2023:

For cleanliness and maintainability of the codebase, in commit 2f351d80 the checks were removed from the above functions and implemented in `getRsv`.

4 Discussion

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

4.1 Economic arbitrage

The in-production expiry of orders will be 120 seconds according to the Bebop team. This opens up a slight avenue of arbitrage behavior, where a user will sign a trade but not execute it, and they will wait 120 seconds. If the market has moved towards their favor, they will execute the trade and make guaranteed profit. Otherwise, they do not submit the order, and they do not take on any loss. While this is not a security issue, we thought it was an interesting observation to note about the mechanism/design of the protocol.

4.2 Maker and taker checks

There are indeed checks to try to thwart the arbitrage behaviour mentioned in the previous discussion point using specific checks:

```
function assertAndInvalidateAggregateOrder(
    AggregateOrder memory order,
    bytes memory takerSig,
    Signature[] memory makerSigs
) public override returns (bytes32) {
    ...
    require(msg.sender ≠ taker, “Taker cannot submit order”);
    ...
}
```

```
function assertAndInvalidateMakerOrders(
    AggregateOrder memory order,
    Signature[] memory makerSigs
) private {
    ...
    require(msg.sender ≠ maker_address, “No maker can submit order”);
    ...
```

```
}
```

However, these checks are superficial and can be easily bypassed by sending order details to another EOA to be submitted.

4.3 EIP1271 code execution

If the orders are settled using the EIP1271 system, a market maker can execute any code in the `isValidSignature` callback. This allows for schemes where a market maker checks oracles/prices during the callback and reverts the entire transaction if they are not guaranteed a profit. This callback is also not called with a gas limit, so it could be used maliciously to burn a lot of gas or even perform other dex arbitrages that are not profitable due to gas fees.

4.4 Reentrancy

There are no reentrancy guards in any of the functions, and there are three possible reentrancy spots.

```
function validateMakerSignature( ... )
{
    } else if (signature.signatureType == SignatureType.EIP1271) {
        require(IERC1271(signature.walletAddress).isValidSignature(hash,
    signature.signatureBytes) == EIP1271_MAGICVALUE, “Invalid Maker EIP
    1271 Signature”);
}
```

```
SettleAggregateOrder( ... )
{
    // transfer each of those tokens to the corresponding maker
    for (uint k = 0; k < takerTokensLength; k++){
        IERC20(address(order.taker_tokens[i][k])).safeTransferFrom(
            order.taker_address,
            order.maker_addresses[i],
            order.taker_amounts[i][k]
        );
    }
}
```

```
function makerTransferFunds(
    address from,
    address to,
    uint256 quantity,
    address token
) private returns (bool) {
    IERC20(token).safeTransferFrom(from, to, quantity);
    return true;
}
```

We did not however note any reentrancy-related security issues, though it is important to keep in mind these possible reentrancy areas in future development.

5 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.

5.1 File: bebop_aggregation_contract

Function: `assertAndInvalidateAggregateOrder(order, takerSig, makerSigs)`

Intended behavior

- Constructing and validating the partial orders of the aggregator.

Branches and code coverage

Intended branches:

- Ensure that the `order` is indeed the order that the `taker` wanted (done through `ecrecover(hash_of_order, V, R ,S)` where `V R S` are from `takerSig`).
  - ☑ Test coverage
- Ensure that `taker == order.taker`. This is performed in code.
  - ☑ Test coverage
- Shouldn't be `public`. Will be mitigated as part of the remediation.
  - ☑ Test coverage

Negative behavior:

- Shouldn't allow performing orders that `takers` or `makers` did not sign.
  - ☐ Negative test?
- Shouldn't allow performing `expired` orders. Should be covered by the code.
  - ☐ Negative test?
- Assume that `takerSig` cannot be crafted on behalf of arbitrary users, since they may have previously approved the transaction.
  - ☐ Negative test?
- Ensure that `msg.sender` $\neq$ `taker` and `msg.sender` $\neq$ `maker`. Covered by code.
  - ☑ Negative test?

Preconditions

- Assumes the `takersig` and `makerSigs` are valid (for this specific order) and that they cannot be duplicated.
- Assumes the `order` is still valid (`expiry > timestamp`).

Inputs

- `Signature makerSigs`:
  - **Control**: Full control; passed on to `assertAndInvalidateMakerOrders`.
  - **Authorization**: checks are performed at the level of `assertAndInvalidateMakerOrders`.
  - **Impact**: Could impact the authorization.
- `bytes takerSig`:
  - **Control**: Full control; assumes that it cannot be maliciously crafted on behalf of other users.
  - **Authorization**: Checks that it matches the order's taker after recovering the address. (who would have signed the order).
  - **Impact**: Could impact the authorization.
- `AggregateOrder order`:
  - **Control**: Full control; there's checks on whether `taker` $\neq$ `order.taker`.
  - **Authorization**: Any order can be supplied, assuming that it's backed up by the `takerSig` and `makerSigs`.
  - **Impact**: Impacts the order flow.

External call analysis

- `hashAggregateOrder(order)`:
  - **What is controllable?** The `order` - it hashes it and returns the `bytes32` of its hash.
  - **If return value controllable, how is it used and how can it go wrong?** returns the hash of the order, will eventually fail if the taker & maker did not approve the order.
  - **What happens if it reverts, reenters, or does other unusual control flow?** n/a.
- `getRsv(takerSig)`:
  - **What is controllable?** Decomposes the signature into `r,s,v` such that the signature can be recovered.
  - **If return value controllable, how is it used and how can it go wrong?** `r,s,v` are used to recover the address of the original signer, such that it can later be used as a means of validating certain actions.

  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot reenter; it could fail silently if the length of `takerSig` $\neq$ `65`. We wrote a finding about this, and it should be mitigated.
- `ecrecover(h, V, R, S);`:
  - **What is controllable?** `V, R, S` are retrieved from the `takerSig`. It's important that the `h` (hash) cannot be arbitrarily written, such that no crafting can be performed.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the original signer of the specific signature. That's supposed to match the `takerSig`, since that's the address of the `taker`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this fails, it should return `0x0` address, so there could be a check against that.
- `assertAndInvalidateMakerOrders(order, makerSigs)`: Detailed below.

Function: `assertAndInvalidateMakerOrders(order, makerSigs)`

Intended behavior

- Should invalidate the maker orders given by the `assertAndInvalidateAggregateOrder` function.

Branches and code coverage

Intended branches:

- Assure that each individual partial order is approved by the subsequent `makeraddress[i]` (performed in `validateMakerSignature`).
  - ☑ Test coverage
- Assure that none of the `makeraddresses[i]` are `msg.sender`. Covered by code, not tested.
  - ☑ Test coverage
- Assure all arrays' lengths match (one misses - `order.maker_addresses`)
  - ☐ Test coverage

Negative behavior:

- The entire order should no longer be performable after (cannot be replayed).
  - ☐ Negative test?
- The created partial order should match the user's projections in terms of every aspect (the tokens exchanged, their amounts, etc.). Covered through the signing of a order, but not tested.
  - ☐ Negative test?

- `order.maker_addresses` shouldn't be different in length than the other arrays (currently not checked).
  - ☐ Negative test?

Preconditions

- Assumes that the `taker` has previously approved the entire order (done at the level of the function that calls this).

Inputs

- `Signature makerSigs`:
  - **Control**: Full control; it's basically passed as a param from the previous function.
  - **Authorization**: After they are recovered, it should match the current order index's `maker` address.
  - **Impact**: Could impact the authorization.
- `AggregateOrder order`:
  - **Control**: Full control.
  - **Authorization**: There should be checks that for each individual order index, the `maker` signed that particular `partial` order.
  - **Impact**: Impacts the order flow.

Function call analysis

- `invalidateOrder(maker_address, order.maker_nonces[i]);`:
  - **What is controllable?** `maker_address`, and the nonce.
  - **If return value controllable, how is it used and how can it go wrong?** This function essentially marks down the specific nonce after it's been used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, it means that the particular nonce has been used and can no longer be performed.
- `validateMakerSignature(maker_address, partial_hash, makerSig)`: Detailed below.
- `hashPartialOrder(partial_order)`:
  - **What is controllable?** To some extent, the `partial_order` since its crafted as an index of the larger order.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the hash that will be validated.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `validateMakerSignature(maker_address, partial_hash, makerSig)`

Intended behavior

- Validate that the `maker` has a valid signature in the partial order that belongs to them and that is to be performed.

Branches and code coverage

Intended branches:

- Ensure that the `signatureType` really is what it says it is. Ensured in code.
  ☑ Test coverage

Negative behavior:

- Shouldn't allow the arbitrary and unchecked `signature.walletAddress` to be passed as `IERC1271` param. This has been mitigated as part of the audit.
  ☐ Negative test?
- Shouldn't allow `signatureTypes` that are not supported by the protocol. Currently enforced in code.
  ☑ Negative test?
- Shouldn't allow using an arbitrary `hash`. In this case though, `partial_hash` is not arbitrary and is constructed based on the partial order.
  ☑ Negative test?

Preconditions

- Assumes that it's called from a function that performs checks on the params beforehand, especially on the supplied `hash`, which should not be controllable in any way.
- Assumes that the maker has validated the partial order (through `partial_hash`).

Inputs

- `maker_address`:
  - **Control**: Full control.
  - **Authorization**: Checks whether `maker_address` is the `signer` of the `partial_hash` or if the `orderSignerRegistry[maker_address][signer]` is valid. This would have needed to be approved by the `maker_address` in the first place.
  - **Impact**: Could impact the authorization and the order flow.
- `hash`:
  - **Control**: Full control if called as a public function, though that's just a view; otherwise, **partial** control, since it's constructed from the `partial order`.

  - **Authorization**: There's a check that the `recovered` address from the hash and `v,r,s` do match the `maker_address` or the `orderSignerRegistry[maker_address][signer]`.
  - **Impact**: Could impact the authorization.
- `signature`:
  - **Control**: Full control (it's the `makerSignature`).
  - **Authorization**: It's checked that the `makerSignature` really belongs to the `maker` at hand.
  - **Impact**: Could impact the authorization.

Function call analysis

- `IERC1271(signature.walletAddress).isValidSignature(hash, signature.signatureBytes)`:
  - **What is controllable?** `(signature.walletAddress)` and `signatureBytes`
  - **If return value controllable, how is it used and how can it go wrong?** can be controlled if the `walletAddress` is malicious, it should be removed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** It can reenter in case of malicious `walletAddress`, so it should be removed.

Function: `SettleAggregateOrder(AggregateOrder order, bytes takerSig, Signature[] makerSigs)`

Intended behavior

- Should perform the settlement of an aggregate order, executed between the individual `taker` and one or multiple `makers`.

Branches and code coverage

Intended branches:

- Ensure that all partial orders of the aggregate order have been performed.
  ☐ Test coverage
- Assure the transfer of `taker_token` (for each one) is performed and that the `maker`'s balances increase accordingly.
  ☐ Test coverage
- Assure that the balances of the `taker` (for each token) increases by what was decreased from the maker.
  ☐ Test coverage
- Assure that the balances of the `makers` decrease by the `maker_amount`.
  ☐ Test coverage

Negative behavior:

- Should not aggregate orders that have already been settled.
  ☑ Negative test?
- Should not aggregate expired order.
  ☐ Negative test?
- Should not transfer more tokens than intended.
  ☐ Negative test?

Preconditions

- Assumes that each partial order, as well as the entire aggregate order, have been signed beforehand (by the taker and makers).

Inputs

- `makerSigs`:
  - **Control**: Full control.
  - **Authorization**: They are individually validated in `assertAndInvalidateAggregateOrder`.
  - **Impact**: Could impact the authorization.
- `takerSig`:
  - **Control**: Full control.
  - **Authorization**: Validated in `assertAndInvalidateAggregateOrder`.
  - **Impact**: Could impact the authorization.
- `order`:
  - **Control**: Full control.
  - **Authorization**: Hashed and validated within `assertAndInvalidateAggregateOrder`.
  - **Impact**: Could impact the order flow.

Function call analysis

- `IERC20(address(order.taker_tokens[i][k])).safeTransferFrom(order.taker_address, order.maker_addresses[i],order.taker_amounts[i][k])`:
  - **What is controllable?** Every param is.
  - **If return value 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?** Presumably everything has been validated beforehand, so the transfers should happen according to the previously validated order. It shouldn't revert; it can reenter since it's calling the `safeTransferFrom` on a param, but

that's not expected as the `tokens` are supposed to be whitelisted by the `maker`.

- `makerTransferFunds(order.maker_addresses[i], order.receiver, order.maker_amounts[i][j], order.maker_tokens[i][j])`:
  - **What is controllable?** Every param is.
  - **If return value 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?** Presumably everything has been validated beforehand, so the transfers should happen according to the previously validated order. It shouldn't revert; it can reenter since it's calling the `safeTransferFrom` on an arbitrary address, but that's not expected as the `tokens` are supposed to be whitelisted by the `maker`.
- `assertAndInvalidateAggregateOrder(order, takerSig, makerSigs)`:
  - **What is controllable?** Every param is.
  - **If return value 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 should revert if the signatures and/or order does not match with the signatures; moreover, it should mark the order as performed after the function call ends.

Function: `registerAllowedOrderSigner(address signer, bool allowed)`

Intended behavior

- Should set an address as allowed to sign for a order transaction on behalf of a maker.

Branches and code coverage

Intended branches:

- Allow users to change the state whenever they want (to disallow further signature uses).
  ☑ Test coverage
- Change the state of the `orderSignerRegistry` for a specific `msg.sender` and `signer` pair.
  ☑ Test coverage

Negative behavior:

- Shouldn't allow changing an arbitrary address state. (Impossible since it's changed on `msg.sender`.)
  ☑ Negative test?

Preconditions

- Assumes that the `msg.sender` is a maker.

Inputs

- `allowed`:
  - **Control**: Full control.
  - **Authorization**: Changes the state of the mapping, on behalf of the `msg.sender`.
  - **Impact**: Impacts the signer authorization.
- `signer`:
  - **Control**: Full control.
  - **Authorization**: N/A; it's basically on behalf of `msg.sender`, so it wouldn't matter.
  - **Impact**: Impacts the signer authorization.

Function call analysis

N/A.

6 Audit Results

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

During our audit, we discovered five findings. Of these, one was critical, one was of medium risk, and the remaining three were of low risk. Bebop acknowledged all findings and implemented fixes.

6.1 Disclaimers

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 additional code added to the assessed 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 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.