Report

v. 1.0

Customer

Bebop

Smart Contract Audit

Bebop

4th April 2024

Report prepared by

ABDK Consulting

Contents

1 Changelog

#	Date	Author	Description
0.1	04.04.24	A. Zveryanskaya	Initial Draft
0.2	04.04.24	A. Zveryanskaya	Minor revision
1.0	04.04.24	A. Zveryanskaya	Release

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting (Mikhail Vladimirov and Dmitry Khovratovich) at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

Bebop is a suite of decentralized trading products. Our bebop.xyz app offers easy and seamless trading to both beginner and savvy traders. Our APIs and SDK boast powerful and performant tools for any protocol, dApp or DAO trade execution, as well as for liquidity providers and solvers.

3 Project scope

We were asked to review:

- Original Code
- Code with Fixes

Files:

/			
	BebopSettlement.sol		
base/			
	BebopPartner.sol	BebopSigning.sol	BebopTransfer.sol
	Errors.sol		
libs/			
	Commands.sol	Order.sol	Signature.sol
	Transfer.sol		
libs/common/			
	BytesLib.sol	SafeCast160.sol	

4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check if code actually does what it is supposed to do, if that algorithms are optimal and correct, and if proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Minor issues** contain code style, best practices and other recommendations.

5 Our findings

We found 1 major, and a few less important issues.

	Info	Fixed
Major	0	1
Moderate	5	2

Fixed 3 out of 8 issues

6 Major Issues

CVF-4 FIXED

- **Category** Suboptimal
- **Source** Signature.sol

Related code:

```
69 v := shr(248, calldataload(add(sig.offset, 64)))
```

The ‘getRsv‘ function of the ‘Signature‘ contract extracts the ‘r‘, ‘s‘, and ‘v‘ components from a signature in Ethereum:

```
    function getRsv(bytes calldata sig) internal pure returns (
       ↪ bytes32, bytes32, uint8){
        if(sig.length != 65) revert InvalidSignatureLength();
        bytes32 r;
        bytes32 s;
        uint8 v;
        assembly {
            r := calldataload(sig.offset)
            s := calldataload(add(sig.offset, 32))
            v := shr(248, calldataload(add(sig.offset, 64)))
        }
          // ... rest of the function logic
    }
```

The issue is with the way the recovery identifier ‘v‘ is retrieved from the signature data. The bit shifting is unnecessarily complex in this case because the ‘v‘ value has only 1 byte and is always located in the last byte of the signature. Therefore, the same result with less overhead can be achieved by directly loading only that byte (instead of 32 bytes).

Recommendation Replace the bit-shifting operation with a direct byte load to save one bit shift:

```
v := calldataload(add(sig.offset, 33))
```

7 Moderate Issues

CVF-1. INFO

- **Category** Flaw
- **Source** BebopTransfer.sol

In the BebopTransfer abstract contract, the function _transferTakerTokens() is utilized during the “Many-to-One” trade to transfer the taker’s tokens. It receives the ‘order‘ parameter which contains all information about the order including the taker_amounts array. However, under the ‘NATIVE_TRANSFER‘ command condition, the below check

```
159 } else if (command == Commands.NATIVE_TRANSFER) {
160     if (msg.value != order.taker_amounts[i]) revert
        ↪ NotEnoughNativeToken();
```

only verifies the amount of an individual value of the given array index. The check does not provide much value as, due to the “Many-to-One” type of trade, there can be several ‘NATIVE_TRANSFER‘ commands in one MultiOrder making it possible for the attacker to steal ETH from the contract by sending it once and withdrawing the same amounts multiple times in one transaction (if the contract holds any extra ETH).

Pre-conditions
– The ‘BebopSettlement‘ contract holds at least 20 ETH.
– The attacker needs to be able to create and execute Many-to-One MultiOrders.

Attack

1. To create a Many-to-One MultiOrder, attacker crafts an ‘order‘ with two ‘NATIVE_TRANSFER‘ commands and sets the corresponding entries in the ‘taker_amounts‘ array of the ‘order‘ parameter to ‘[10 ETH, 10 ETH]‘.
2. Attacker initiates a transaction calling ‘swapMulti()‘ with the crafted ‘order‘ and 10 ETH as ‘msg.value‘ (matching *only* *the first value* in ‘taker_amounts‘).
3. The contract executes the ‘_transferTakerTokens()‘ function for each ‘NATIVE_TRANSFER‘ command in the ‘order‘.
4. Due to the flawed check, the ‘msg.value‘ (10 ETH) is enough to pass the verification for both transfers (of 10 ETH each).
5. Attacker successfully receives a total of 20 ETH from the contract in a single MultiOrder transaction, effectively stealing 10 ETH.

Recommendation Consider calculating the total sum of ‘takerAmounts[]‘ and matching ‘msg.value‘ against that value.

Client Comment. Contract doesn’t hold any tokens or ETH, so it’s not a problem. And if there is somehow some tokenseth in the contract, then anyone can sweep it using function swapSingleFromContract. Nevertheless, the check has been removed.

CVF-2, 3. FIXED

- **Category** Suboptimal
- **Source** Order.sol, Signature.sol

Related code:

88
```
///                                                 1x: ETH_SIGN
```

45
```
///                 |                 1x: ETH_SIGN
```

The comments for extracting signature types in Signature.sol and Order.sol:

```
     ///+---+----- signature type
    ///            00: EIP-712
    ///            01: EIP-1271
    ///            1x: ETH_SIGN
```

Having multiple valid codes (‘10‘ and ‘11‘) for the same signature type ‘ETH_SIGN‘ is unnecessary and could lead to misinterpretations. If another signature type needs to be added in the future and requires a binary value starting with ‘1‘, it might conflict with the current broad definition of ‘ETH_SIGN‘. The issue appears in two instances: ‘Signature.extractMakerFlags()‘ and ‘Order.extractSignatureType()‘.

Recommendation Consider choosing a single, fixed binary value to represent the ‘ETH_SIGN‘ signature type. For example, use ‘10‘ and reserve unused value ‘11‘ for future use. Also, make sure to document the chosen encoding scheme changes.

CVF-5, 6 INFO

- **Category** Suboptimal
- **Source** BebopTransfer.sol

Related code:

167
```
if (batchTransferDetails.length > 0){
    assembly {mstore(batchTransferDetails, sub(mload(
       ↪ batchTransferDetails), 1))}
```

241
```
assembly {mstore(batchTransferDetails, sub(mload(
   ↪ batchTransferDetails), 1))}
```

The functions _transferTakerTokens() and _transferMakerTokens() in the BebopTransfer contract iterate through the taker_tokens and maker_tokens arrays to process each token transfer. In _transferTakerTokens(), if a PERMIT2 command is encountered, a new element is added to the batchTransferDetails array for each remaining token in the loop. A similar functionality occurs in _transferMakerTokens() with batch transfers. However, resizing an array every iteration is suboptimal and expensive in terms of gas. Solidity needs to allocate new memory, copy elements, and potentially deallocate the old array. This can make the overall transaction more expensive and impact the usability of the contract.

Recommendation Consider counting the number of Permit2 transfers in a local variable and then updating the array length once after the loop.

Client Comment. The current version is more gas optimal in terms of the likelihood of using the contract. Because in your case you will have to spend gas on calculating the length of the array for each trade. And in the current code, we need to shorten the array only if the trade contains a mix of Permit2 and another transfer, which cannot happen when using a standard Bebop flow.

CVF-7 INFO

- **Category** Suboptimal
- **Source** BebopTransfer.sol

Related code:

```
54 address tokenAddress = order.taker_tokens[0][0];
```

In ‘BebopTransfer._getAggregateOrderInfo()‘ the function variable address tokenAddress stores the value of order.taker_tokens[0][0].

However, the transaction will revert every time ‘order‘.**taker_tokens**[**0**][**0**]‘** is empty. This happens because Solidity does not handle null values and an attempt to access a non-existent element leads to a revert.

Recommendation Either implement checks for empty arrays or consider removing the ‘address tokenAddress‘ variable from the function.

```
require(order.taker_tokens.length != 0, "empty array");
```

or

```
if(order.taker_tokens.length == 0) revert EmptyArray();
```

Client Comment. The array is not expected to be empty, since it contains tokens for the trade.

CVF-8 INFO

- **Category** Unclear behavior
- **Source** BebopTransfer.sol

Related code:

178
```
encoded = abi.encodePacked(encoded, keccak256(abi.encodePacked(
   ↪ _nested_array[i])));
```

187
```
encoded = abi.encodePacked(encoded, keccak256(abi.encodePacked(
   ↪ _nested_array[i])));
```

The functions '_encodeTightlyPackedNestedInt()' and '_encodeTightlyPackedNested()' in the 'BebopSigning' contract have the same functionality for packing two-dimensional arrays (BebopSigning.sol):

```
    uint nested_array_length = _nested_array.length;
    for (uint i = 0; i < nested_array_length; i++) {
        encoded = abi.encodePacked(encoded, keccak256(abi.
           ↪ encodePacked(_nested_array[i])));
    }
    return encoded;
```

The current implementation creates a new encoded array on every loop iteration using 'abiėncodePacked()'. However, each iteration allocates fresh memory to hold the concatenated data, and previous data is copied into the new array repeatedly. This is redundant and consumes significant amounts of gas, especially for larger nested arrays.

Recommendation To optimize the functions '_encodeTightlyPackedNestedInt()' and '_encodeTightlyPackedNested()' consider avoiding redundant array allocations and data copies. For example, consider redesigning them as follows:

1. Pre-calculate the total length for a resulting array.
2. Allocate the encoded bytes array with the calculated length once, before the loop, to avoid multiple allocations.
3. Append all the arrays directly to the pre-allocated encoded array using ' abiėncodePacked , eliminating redundant copies.

8 Recommendations

CVF-9 INFO

- **Category** Suboptimal
- **Source** BytesLib.sol

Description This function is overcomplicated and suboptimal.

Recommendation Here is how it could be simplified and optimized.

```
5 function slice(bytes memory _bytes, uint256 _start, uint256 _length)
    ↪  internal pure returns (bytes memory) {
```

CVF-10 INFO

- **Category** Suboptimal
- **Source** BytesLib.sol

Description The expression “add(lengthmod (mul(0×20, iszero(lengthmod)))” is calculated twice.

Recommendation Consider calculating once and reusing.

```
27 let mc := add(add(tempBytes, lengthmod), mul(0x20, iszero(lengthmod)
    ↪ ))
```

```
32     let cc := add(add(add(_bytes, lengthmod), mul(0x20, iszero(
        ↪ lengthmod))), _start)
```

CVF-11 INFO

- **Category** Suboptimal
- **Source** SafeCast160.sol

Recommendation This function could be simplified as: function toUint160(uint256 value) internal pure returns (uint160 result) { if ((result = uint160(value)) != value) revert UnsafeCast(); }

```
10 function toUint160(uint256 value) internal pure returns (uint160) {
       if (value > type(uint160).max) revert UnsafeCast();
       return uint160(value);
   }
```

CVF-12 INFO

- **Category** Bad datatype
- **Source** Order.sol

Recommendation The type for these fields should be “IERC20”.

```
15 address taker_token;
   address maker_token;
```

CVF-13 INFO

- **Category** Bad datatype
- **Source** Order.sol

Recommendation The type for these fields should be “IERC20[]”.

```
31 address[] taker_tokens;
   address[] maker_tokens;
```

CVF-14 INFO

- **Category** Suboptimal
- **Source** Order.sol

Recommendation It would be more efficient to use a single array of structs with four fields, rather than four parallel arrays. This would also make the length checks unnecessary.

Client Comment *Arrays can be of different lengths because the number of maker and taker tokens can be different.*

```
31 address[] taker_tokens;
   address[] maker_tokens;
   uint256[] taker_amounts;
   uint256[] maker_amounts;
```

CVF-15 INFO

- **Category** Suboptimal
- **Source** Order.sol

Recommendation It would be more efficient to use a single array of structs with six fields, rather than six parallel arrays. This would also make the length checks unnecessary.

Client Comment *Arrays can be of different lengths because the number of maker and taker tokens can be different.*

```
44 address[] maker_addresses;
   uint256[] maker_nonces;
   address[][] taker_tokens;
   address[][] maker_tokens;
   uint256[][] taker_amounts;
   uint256[][] maker_amounts;
```

CVF-16 INFO

- **Category** Bad datatype
- **Source** Order.sol

Recommendation The type for these fields should be “IERC20[][]”.

```
46 address[][] taker_tokens;
   address[][] maker_tokens;
```

CVF-17 INFO

- **Category** Suboptimal
- **Source** Order.sol

Recommendation It would be more efficient to use a single array of structs with four fields, rather than four parallel arrays. This would also make the length checks unnecessary.

Client Comment *Arrays can be of different lengths because the number of maker and taker tokens can be different.*

```
46 address[][] taker_tokens;
   address[][] maker_tokens;
   uint256[][] taker_amounts;
   uint256[][] maker_amounts;
```

CVF-18 INFO

- **Category** Suboptimal
- **Source** Order.sol

Recommendation In case ETHSIGN would always be encoded as 10, it would be possible to simplify this code like this: signatureType = Signature.Type(flags & 0×3);”

Client Comment *Arrays can be of different lengths because the number of maker and taker tokens can be different.*

```
90 if (flags & 0x02 != 0) {
       signatureType = Signature.Type.ETHSIGN;
   } else if (flags & 0x01 != 0) {
       signatureType = Signature.Type.EIP1271;
   } else {
       signatureType = Signature.Type.EIP712;
   }
```

CVF-19 FIXED

- **Category** Suboptimal
- **Source** Order.sol

Recommendation This could be simplified as: partnerId = uint64(flags » 64);

```
101 partnerId = uint64((flags &
    0x0000000000000000000000000000000FFFFFFFFFFFFFFF0000000000000000)
       ↪ >> 64);
```

CVF-20 FIXED

- **Category** Suboptimal
- **Source** Order.sol

Recommendation This could be simplified as: partnerId = uint64(flags » 64);

```
105 partnerId = uint64((flags &
    0x0000000000000000000000000000000FFFFFFFFFFFFFFF0000000000000000)
       ↪ >> 64);
```

CVF-21 INFO

- **Category** Suboptimal
- **Source** Transfer.sol

Recommendation It would be more efficient to use a single array of structs with two fields, rather than two parallel arrays. This would also make the length check unnecessary.

```
21 uint256[][] makerAmounts;
   uint256[] makerNonces;
```

CVF-22 INFO

- **Category** Bad datatype
- **Source** Transfer.sol

Recommendation The type for this field should be “IERC20”.

```
37 address token;
```

CVF-23 INFO

- **Category** Procedural
- **Source** Commands.sol

Recommendation Consider replacing this set of constants with an enum.

```
8  bytes1 internal constant SIMPLE_TRANSFER = 0x00; // simple transfer
      ↪ with standard transferFrom
   bytes1 internal constant PERMIT2_TRANSFER = 0x01; // transfer using
      ↪ Permit2.transfer
10 bytes1 internal constant CALL_PERMIT_THEN_TRANSFER = 0x02; // call
      ↪ permit then standard transferFrom
   bytes1 internal constant CALL_PERMIT2_THEN_TRANSFER = 0x03; // call
      ↪ Permit2.permit then Permit2.transfer
   bytes1 internal constant NATIVE_TRANSFER = 0x04; // wrap/unwrap
      ↪ native token and transfer
   bytes1 internal constant TRANSFER_TO_CONTRACT = 0x07; // transfer to
      ↪  bebop contract
   bytes1 internal constant TRANSFER_FROM_CONTRACT = 0x08; // transfer
      ↪ from bebop contract
```

CVF-24 FIXED

- **Category** Suboptimal
- **Source** Signature.sol

Recommendation In case ETHSIGN would always be encoded as 10, it would be possible to simplify this code like this: signatureType = Type(flags & 0×3);

```
50 if (flags & 0x02 != 0) {
       signatureType = Signature.Type.ETHSIGN;
   } else if (flags & 0x01 != 0) {
       signatureType = Signature.Type.EIP1271;
   } else {
       signatureType = Signature.Type.EIP712;
   }
```

CVF-25 FIXED

- **Category** Suboptimal
- **Source** Signature.sol

Recommendation The “Signature.” prefix is redundant.

```
51 signatureType = Signature.Type.ETHSIGN;
```

```
53 signatureType = Signature.Type.EIP1271;
```

```
55 signatureType = Signature.Type.EIP712;
```

CVF-26 INFO

- **Category** Bad datatype
- **Source** Signature.sol

Recommendation This value should be a named constant.

```
72 if(uint256(s) >
   0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0)
      ↪ revert InvalidSignatureValueS();
```

CVF-27 INFO

- **Category** Suboptimal
- **Source** Errors.sol

Recommendation These errors could be made more useful by adding certain parameters into them.

```
error OrderInvalidSigner();
error InvalidEIP721Signature();
error InvalidEIP1271Signature();
error InvalidETHSIGNSignature();
error InvalidSignatureType();
error InvalidSignatureLength();
error InvalidSignatureValueS();
error InvalidSignatureValueV();
```

```
error InvalidNonce();
error OrderExpired();
error OrdersLengthsMismatch();
error TokensLengthsMismatch();
error CommandsLengthsMismatch();
error InvalidPermit2Commands();
error InvalidCommand();
error InvalidCommandsLength();
error InvalidFlags();
error PartialFillNotSupported();
error UpdatedMakerAmountsTooLow();
```

```
error MakerAmountsLengthsMismatch();
```

```
error NotEnoughNativeToken();
error WrongWrappedTokenAddress();
error FailedToSendNativeToken();
```

```
error InvalidSender();
error ManyToManyNotSupported();
```

```
error InvalidPendingTransfersLength();
error InvalidPermit2TransfersLength();
```

```
error PartnerAlreadyRegistered();
error PartnerFeeTooHigh();
```

CVF-28 INFO

- **Category** Documentation
- **Source** BebopPartner.sol

Description The semantics of the keys in this mapping is unclear.

Recommendation Consider documenting.

```
13 mapping(uint64 => PartnerInfo) public partners;
```

CVF-29 FIXED

- **Category** Suboptimal
- **Source** BebopPartner.sol

Description The expression “partners[0]” is calculated three times.

Recommendation Consider calculating once and reusing.

```
17 partners[0].fee = 0;
   partners[0].beneficiary = address(0);
   partners[0].registered = true;
```

CVF-30 FIXED

- **Category** Unclear behavior
- **Source** BebopPartner.sol

Description These assignments do nothing as default values for storage variable are zero.

```
17 partners[0].fee = 0;
   partners[0].beneficiary = address(0);
```

CVF-31 INFO

- **Category** Suboptimal
- **Source** BebopPartner.sol

Description The expression “partners[partnerId]” is calculated twice.

Recommendation Consider calculating once and reusing.

```
28 if(partners[partnerId].registered) revert PartnerAlreadyRegistered()
       ↪ ;
```

```
31 partners[partnerId] = PartnerInfo(fee, beneficiary, true);
```

CVF-32 INFO

- **Category** Procedural
- **Source** BebopPartner.sol

Recommendation These checks should be made earlier.

```
29 if(fee > HUNDRED_PERCENT) revert PartnerFeeTooHigh();
30 if (beneficiary == address(0)) revert NullBeneficiary();
```

CVF-33 INFO

- **Category** Suboptimal
- **Source** BebopPartner.sol

Description This check is redundant, as it is anyway possible to pass an invalid partner address.

Recommendation Consider removing this check.

```
30 if (beneficiary == address(0)) revert NullBeneficiary();
```

CVF-34 INFO

- **Category** Procedural
- **Source** BebopTransfer.sol

Description We didn't review these files.

```
4 import "../interface/IDaiLikePermit.sol";
  import "../interface/IPermit2.sol";
  import "../interface/IWETH.sol";
```

CVF-35 INFO

- **Category** Bad datatype
- **Source** BebopTransfer.sol

Recommendation The type for this variable should be "IWETH".

```
22 address internal immutable WRAPPED_NATIVE_TOKEN;
```

CVF-36 INFO

- **Category** Bad datatype
- **Source** BebopTransfer.sol

Recommendation The type for this token should be more specific.

```
23 address internal immutable DAI_TOKEN;
```

CVF-37 FIXED

- **Category** Suboptimal
- **Source** BebopTransfer.sol

Recommendation This variable is redundant. Just give a name to the returned value.

```
30 uint256 id;
```

CVF-38 INFO

- **Category** Bad datatype
- **Source** BebopTransfer.sol

Recommendation The type for the “_wrappedNativeToken” argument should be “IWETH”.

```
37 constructor(address _wrappedNativeToken, address _permit2, address
   ↪ _daiAddress) {
```

CVF-39 INFO

- **Category** Bad datatype
- **Source** BebopTransfer.sol

Recommendation The type for the “_permit2” argument should be “IPermit2”.

```
37 constructor(address _wrappedNativeToken, address _permit2, address
   ↪ _daiAddress) {
```

CVF-40 INFO

- **Category** Bad datatype
- **Source** BebopTransfer.sol

Recommendation The type for the “_daiAddress” argument should be more specific.

```
37 constructor(address _wrappedNativeToken, address _permit2, address
   ↪ _daiAddress) {
```

CVF-41 FIXED

- **Category** Procedural
- **Source** BebopTransfer.sol

Description The returned values actually don't have names.

Recommendation Consider naming them.

```
45 /// @return quoteTakerAmount - full taker_amount for One-to-One or
      ↪ One-to-Many trades, for Many-to-One orders it's 0
   /// @return lenInfo - lengths of `pendingTransfers` and `
      ↪ batchTransferDetails` arrays

50 ) internal pure returns (uint, Transfer.LengthsInfo memory){
```

CVF-42 INFO

- **Category** Procedural
- **Source** BebopTransfer.sol

Recommendation This check be done earlier.

```
65 if (tokenAddress != order.taker_tokens[i][j]){
```

CVF-43 FIXED

- **Category** Readability
- **Source** BebopTransfer.sol

Recommendation Should be "else if".

```
72 if (curCommand == Commands.PERMIT2_TRANSFER || curCommand ==
      ↪ Commands.CALL_PERMIT2_THEN_TRANSFER) {
```

CVF-44 INFO

- **Category** Bad datatype
- **Source** BebopTransfer.sol

Recommendation The type for the “token” argument should be “IERC20”.

```
93 address from, address to, address token, uint256 amount, bytes1
   ↪ command, Transfer.Action action, uint64 partnerId
```

CVF-45 INFO

- **Category** Bad datatype
- **Source** BebopTransfer.sol

Recommendation The type for this argument should be “IERC20[]”.

```
188 address[] calldata maker_tokens,
```

CVF-46 INFO

- **Category** Bad datatype
- **Source** BebopTransfer.sol

Recommendation The type for the “tokenAddress” argument should be “IERC20”.

```
326 address takerAddress, address tokenAddress, Signature.
   ↪ PermitSignature calldata takerPermitSignature
```

```
346 address takerAddress, address tokenAddress, Signature.
   ↪ Permit2Signature calldata takerPermit2Signature
```

CVF-47 INFO

- **Category** Bad naming
- **Source** BebopSigning.sol

Recommendation Events are usually named via nouns, such as “OrderSigner”.

```
13 event OrderSignerRegistered(address maker, address signer, bool
   ↪ allowed);
```

CVF-48 INFO

- **Category** Suboptimal
- **Source** BebopSigning.sol

Recommendation Consider using a hash expression instead of a hardcoded value. Solidity compiler is smart enough to precompute hash expressions.

Client Comment *No it's not smart enough, because in that case bytecode increases and there is an error: "Contract code size is 25078 bytes and exceeds 24576 bytes".*

```
18 // bytes4(keccak256("isValidSignature(bytes32,bytes)"))
   bytes4 private constant EIP1271_MAGICVALUE = 0x1626ba7e;

24 /// keccak256(abi.encodePacked(
   ///   "EIP712Domain(string name,string version,uint256 chainId,
      ↪ address verifyingContract)"
   /// ));
   bytes32 private constant EIP712_DOMAIN_TYPEHASH =
   0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f;

30 /// keccak256(abi.encodePacked(
   ///   "AggregateOrder(uint64 partnerId,uint256 expiry,address
      ↪ taker_address,address[] maker_addresses,uint256[] maker_nonces
      ↪ ,address[][] taker_tokens,address[][] maker_tokens,uint256[][]
      ↪  taker_amounts,uint256[][] maker_amounts,address receiver,
      ↪ bytes commands)"
   /// ));
   bytes32 private constant AGGREGATED_ORDER_TYPE_HASH =
   0x3d51b1f10b459dc4622654aaef17e16ff20b62e8761c430cb560da4c77abfb2f;

36 /// keccak256(abi.encodePacked(
   ///   "MultiOrder(uint64 partnerId,uint256 expiry,address
      ↪ taker_address,address maker_address,uint256 maker_nonce,
      ↪ address[] taker_tokens,address[] maker_tokens,uint256[]
      ↪ taker_amounts,uint256[] maker_amounts,address receiver,bytes
      ↪ commands)"
   /// ));
   bytes32 private constant MULTI_ORDER_TYPE_HASH =
40 0xa02532bc5112e1e5e70d8f4d7600f876b48a99c7c65c04bc378eac85ca24df27;
```

(42)

CVF-49 INFO

- **Category** Documentation
- **Source** BebopSigning.sol

Recommendation Consider explaining how this value was calculated, or even better, specify the value as an expression.

```
21 uint256 private constant ETH_SIGN_HASH_PREFIX =
   0x19457468657265756d205369676e6564204d6573736167653a0a333200000000;
```

CVF-50 INFO

- **Category** Documentation
- **Source** BebopSigning.sol

Description The exact semantics of keys and values in these mappings is unclear.

Recommendation Consider documenting.

```
50 mapping(address => mapping(uint256 => uint256)) private
      ↪ makerNonceValidator;
   mapping(address => mapping(address => bool)) private
      ↪ orderSignerRegistry;
```

CVF-51 INFO

- **Category** Procedural
- **Source** BebopSigning.sol

Description This expression is coded twice.

Recommendation Consider extracting into a utility function.

```
56 abi.encode(EIP712_DOMAIN_TYPEHASH, DOMAIN_NAME, DOMAIN_VERSION,
      ↪ block.chainid, address(this))
```

```
66     abi.encode(EIP712_DOMAIN_TYPEHASH, DOMAIN_NAME, DOMAIN_VERSION,
          ↪ block.chainid, address(this))
```

CVF-52 INFO

- **Category** Unclear behavior
- **Source** BebopSigning.sol

Description This event is emitted even if nothing actually changed.

```
75 emit OrderSignerRegistered(msg.sender, signer, allowed);
```

CVF-53 INFO

- **Category** Procedural
- **Source** BebopSettlement.sol

Description We didn't review this file.

```
6 import "./interface/IBebopSettlement.sol";
```

CVF-54 INFO

- **Category** Bad datatype
- **Source** BebopSettlement.sol

Recommendation The type for the "_wrappedNativeToken" argument should be "IWETH".

```
17 constructor(address _wrappedNativeToken, address _permit2, address
   ↪ _daiAddress)
```

CVF-55 INFO

- **Category** Bad datatype
- **Source** BebopSettlement.sol

Recommendation The type for the "_permit2" argument should be "IPermit2".

```
17 constructor(address _wrappedNativeToken, address _permit2, address
   ↪ _daiAddress)
```

CVF-56 INFO

- **Category** Bad datatype
- **Source** BebopSettlement.sol

Recommendation The type for the "_daiAddress" should be more specific.

```
17 constructor(address _wrappedNativeToken, address _permit2, address
   ↪ _daiAddress)
```

CVF-57 INFO

- **Category** Procedural
- **Source** BebopSettlement.sol

Recommendation It is a good practice to put a comment into an empty block to explain why the block is empty.

```
21 receive() external payable {}
```

CVF-58 INFO

- **Category** Suboptimal
- **Source** BebopSettlement.sol

Description This check makes it redundant passing the "taker_address" field.

Recommendation Consider not passing this field, removing this check, and just using "msg.sender" instead to save gas.

```
37  if (msg.sender != order.taker_address) revert InvalidSender();
46  if (msg.sender != order.taker_address) revert InvalidSender();
123 if (msg.sender != order.taker_address) revert InvalidSender();
198 if (msg.sender != order.taker_address) revert InvalidSender();
```

CVF-59 FIXED

- **Category** Procedural
- **Source** BebopSettlement.sol

Description The order of actual arguments and the order of corresponding descriptions are different.

Recommendation Consider reordering either arguments or descriptions.

265
```
/// @param updatedMakerAmount for RFQ-M case maker amount can be
   ↪ improved
/// @param takerTransferCommand Command to indicate how to transfer
   ↪ taker's token
```

271
```
    bytes1 takerTransferCommand,
    uint256 updatedMakerAmount
```

CVF-60 INFO

- **Category** Procedural
- **Source** BebopSettlement.sol

Recommendation Brackets are redundant.

288
```
newMakerAmount = (updatedMakerAmount * filledTakerAmount) / order.
   ↪ taker_amount;
```

CVF-61 INFO

- **Category** Suboptimal
- **Source** BebopSettlement.sol

Description This event contains too little information to really be useful.

Recommendation Consider adding more parameters into it.

296
```
emit BebopOrder(eventId);
```

351
```
emit BebopOrder(eventId);
```

421
```
emit BebopOrder(Order.extractEventId(order.flags));
```

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting