2022-05-opensea-seaport (OpenSea Seaport)

Benchmark Security Review Performed by Bug Hunter

25.06.2025

Table of Contents

About Bug Hunter

Bug Hunter is an automated code-review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code-analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20 000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit:

- Bug Hunter Website
- Bug Hunter Documentation
- Example Reports

To explore more about Truscova, visit:

- Truscova Website
- X
- LinkedIn

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code-review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and are dependent on Bug Hunter's coverage of detector rules. The full list of rules covered by Bug Hunter automated code review is available at https://docs.bughunter.live/.

Executive Summary

Overview and Scope

A codebase from project 2022-05-opensea-seaport was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/code-423n4/2022-05-opensea-seaport
  Commit hash: `6c24d09fc4be9bbecf749e6a7a592c8f7b659405`

Bug Hunter reviewed the following files:

OrderValidator.sol FulfillmentApplier.sol CriteriaResolution.sol BasicOrderFulfiller.sol Seaport.sol ConsiderationBase.sol LowLevelHelpers.sol ZoneInteraction.sol TokenTransferrer.sol AmountDeriver.sol Conduit.sol Verifiers.sol Executor.sol Assertions.sol SignatureVerification.sol OrderFulfiller.sol OrderCombiner.sol Executor.sol Seaport.sol Consideration.sol ReferenceConsideration.sol

Summary of Findings

The uncovered vulnerabilities identified during the security review are summarised in the table below:

Severity	Count
High	1
Medium	1
Low	11

Project Summary

Contact Information

For support, contact us at support@bughunter.live

Project Timeline

Date	Event
25.06.2025	Project submitted
25.06.2025	Security review concluded

Detailed Findings

1. Potential Truncation When Downcasting to uint120

Severity	High
Finding ID	BH-2022-05-opensea-seaport-01
Target	contracts/lib/OrderValidator.sol
Function name	_validateOrderAndUpdateStatus

Description

In the final state update, the function downcasts `filledNumerator + numerator` and `denominator` from `uint256` to `uint120` using:

```
_orderStatus[orderHash].numerator = uint120(filledNumerator + numerator);
_orderStatus[orderHash].denominator = uint120(denominator);
```

If `filledNumerator + numerator` or `denominator` exceed `2**120 - 1`, truncation will silently occur, corrupting the order's fill state. Solidity does not automatically revert on narrowing conversions unless explicitly checked. This is particularly dangerous since these values can be indirectly influenced by external input.

```
function _validateOrderAndUpdateStatus(
        AdvancedOrder memory advancedOrder,
        CriteriaResolver[] memory criteriaResolvers,
        bool revertOnInvalid,
        bytes32[] memory priorOrderHashes
    )
        internal
        returns (
            bytes32 orderHash,
            uint256 newNumerator,
            uint256 newDenominator
        )
    {
        ...
        unchecked {
            _orderStatus[orderHash].isValidated = true;
            _orderStatus[orderHash].isCancelled = false;
            _orderStatus[orderHash].numerator = uint120(filledNumerator +
numerator);
            _orderStatus[orderHash].denominator = uint120(denominator);
        }
        ...
    }
```

Recommendation

Before downcasting, insert validation logic to ensure values are within the valid `uint120` range:

```
require(filledNumerator + numerator <= type(uint120).max, "Overflow in numerator");
require(denominator <= type(uint120).max, "Overflow in denominator");
```

Alternatively, consider keeping the internal representation as `uint256` if high-range values are expected in some edge cases.

2. Improper Validation of Merkle Proofs in Criteria Resolution

Severity	Medium
Finding ID	BH-2022-05-opensea-seaport-02
Target	contracts/lib/CriteriaResolution.sol
Function name	_applyCriteriaResolvers

Description

The `_applyCriteriaResolvers` function attempts to validate that a given token identifier satisfies a collection-level criteria using a Merkle proof (`criteriaResolver.criteriaProof`). However, there is no validation of the structure or length of the `criteriaProof` array prior to calling `_verifyProof`. If `_verifyProof` does not enforce strict proof validation internally (e.g., length, depth, ordering), malformed proofs could bypass the check, resulting in unauthorized tokens being matched to criteria-based orders. This breaks the integrity of 'criteria-based' matching, allowing an attacker to fulfill orders with invalid items.

```
function _applyCriteriaResolvers(
        AdvancedOrder[] memory advancedOrders,
        CriteriaResolver[] memory criteriaResolvers
    ) internal pure {
        ...
        if (identifierOrCriteria != uint256(0)) {
            // Verify identifier inclusion in criteria root using proof.
            _verifyProof(
                criteriaResolver.identifier,
                identifierOrCriteria,
                criteriaResolver.criteriaProof
            );
        }
        ...
    }
```

Recommendation

Ensure `_verifyProof` enforces proper Merkle proof validation — including expected proof length, sorted sibling pairs (if required), and correct final root match. Additionally, consider adding explicit checks in `_applyCriteriaResolvers` to validate that non-zero `criteriaProof` arrays contain expected structural integrity before trusting the output of `_verifyProof`.

3. Malformed ABI Encoding in `_name()` Function Return

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-03
Target	contracts/Seaport.sol
Function name	_name

Description

The `_name()` function in `Seaport.sol` returns improperly formatted data due to incorrect ABI encoding in inline assembly. This causes the function to return a `string` value that contains dirty or misaligned memory, which may confuse frontends or interfaces expecting clean, ABI-compliant return values. This breaks compatibility with tools or interfaces that rely on metadata such as token name.

```
function _name() internal pure override returns (string memory) {
    assembly {
        mstore(0, 0x20)
        mstore(0x27, 0x07536561706f7274)
        return(0, 0x60)
    }
}
```

Recommendation

Ensure the memory layout for returning a `string` matches Solidity's ABI specification: offset at 0x00, length at 0x20, and properly padded UTF-8 string data at 0x40. Return a total of 0x60 bytes.

4. Insufficient Validation of Dynamic Calldata Offsets in ERC1155 Batch Transfer

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-04
Target	contracts/lib/TokenTransferrer.sol
Function name	_performERC1155BatchTransfers

Description

The `_performERC1155BatchTransfers()` function uses inline assembly to optimize ERC1155 batch transfers, but it fails to fully validate the calldata layout for nested dynamic arrays (`ids` and `amounts`). While some basic encoding checks are present (matching lengths, offsets), it does not ensure that `ids.length` and `amounts.length` point to valid, in-bounds regions of calldata. This allows a malicious caller to construct calldata that overflows memory, triggers excessive gas usage, or copies out-of-bounds data — potentially leading to a denial-of-service condition or undefined behavior during transfer.

```
function _performERC1155BatchTransfers(
        ConduitBatch1155Transfer[] calldata batchTransfers
    ) internal {
        assembly {
            let len := batchTransfers.length
            ... // optimized transfer loop
            calldatacopy(...)
            ...
        }
    }
```

Recommendation

Add strict validation to ensure that: - `ids.length` and `amounts.length` are below a reasonable cap (e.g. 1000) - Calldata regions derived from dynamic offsets fit within `calldatasize()` - `idsAndAmountsSize` and `transferDataSize` do not exceed calldata length Alternatively, consider using Solidity's safer `abi.decode` path with bounds checks, or refactor the assembly logic to add preconditions before `calldatacopy` .

5. Unsafe Hardcoded Offset Checks Without Calldata Bounds Validation

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-05
Target	contracts/lib/Assertions.sol
Function name	_assertValidBasicOrderParameterOffsets

Description

The `_assertValidBasicOrderParameterOffsets` function validates the ABI layout of a `BasicOrderParameters` struct using hardcoded offsets and inline assembly. However, it performs multiple unguarded `calldataload` operations without first checking that calldata is long enough. If calldata is too short, this may cause out-of-bounds reads or silently fail. Moreover, it uses user-controlled values (e.g. `additionalRecipients.length`) to compute expected offsets without overflow protections, which may be manipulated to break downstream assumptions. This creates potential denial-of-service and logic corruption risks.

```
function _assertValidBasicOrderParameterOffsets() internal pure {
    assembly {
        validOffsets := and(
            eq(calldataload(BasicOrder_parameters_cdPtr),
BasicOrder_parameters_ptr),
            eq(calldataload(BasicOrder_additionalRecipients_head_cdPtr),
BasicOrder_additionalRecipients_head_ptr)
        )
        validOffsets := and(
            validOffsets,
            eq(
                calldataload(BasicOrder_signature_cdPtr),
                add(BasicOrder_signature_ptr,
mul(calldataload(BasicOrder_additionalRecipients_length_cdPtr),
AdditionalRecipients_size))
            )
        )
    }
    if (!validOffsets) {
        revert InvalidBasicOrderParameterEncoding();
    }
}
```

Recommendation

Before reading any calldata offset via assembly, validate that `calldatasize()` is sufficient. Use Solidity-level decoding when possible. Validate that all user-influenced fields used for pointer arithmetic (like `additionalRecipients.length`) are capped and safe. Consider replacing hardcoded ABI offset checks with safer, compiler-driven struct validations.

6. Incorrect Magic Value Extraction in `_doesNotMatchMagic`

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-06
Target	contracts/lib/LowLevelHelpers.sol
Function name	_doesNotMatchMagic

Description

The `_doesNotMatchMagic` function attempts to compare a returned `bytes4` magic value by directly loading a full word (`mload(0)`) from returndata. However, this approach incorrectly includes trailing bytes beyond the first 4. If the return data is `0x12345678` followed by non-zero padding, the comparison will fail even though the actual `bytes4` value is correct. This leads to false negatives in interface compliance checks like EIP-1271 or ERC721 receiver validation.

```
function _doesNotMatchMagic(bytes4 expected) internal pure returns (bool) {
    bytes4 result;
    assembly {
        if eq(returndatasize(), OneWord) {
            returndatacopy(0, 0, OneWord)
            result := mload(0) // incorrect, full 32 bytes loaded
        }
    }
    return result != expected;
}
```

Recommendation

Update the assembly block to shift the loaded value so only the first 4 bytes are compared: `result := shr(224, mload(0))`. This ensures proper alignment for `bytes4` values.

7. Missing 's' Value Canonical Check in ECDSA Signature Verification

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-07
Target	contracts/lib/SignatureVerification.sol
Function name	_assertValidSignature

Description

The `_assertValidSignature` function fails to enforce that the `s` value in an ECDSA signature lies in the lower half of the secp256k1 curve. This opens the door to signature malleability, where multiple valid signatures can be generated for the same message. Such malleable signatures can be used to undermine protocols relying on signature uniqueness, replay protection, or deterministic signing.

```
function _assertValidSignature(...) {...}
```

Recommendation

Add a check that `s` <= secp256k1n / 2 to ensure signatures are canonical. Use the following condition: `require(uint256(s) <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0, 'Invalid signature s value');`.

8. Misleading Error Message on Reuse of Fully Filled Basic Order

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-08
Target	contracts/lib/Verifiers.sol
Function name	_verifyOrderStatus

Description

When attempting to execute a basic order that has already been fully filled, the function `_verifyOrderStatus()` reverts with `OrderPartiallyFilled(orderHash)`, which inaccurately describes the issue. The order has been completely filled, not partially filled. This leads to confusion for developers and integrators who rely on revert messages for debugging or UI messaging.

```
function _verifyOrderStatus(
        bytes32 orderHash,
        OrderStatus memory orderStatus,
        bool onlyAllowUnused,
        bool revertOnInvalid
    ) internal pure returns (bool valid) {
        // Ensure that the order has not been cancelled.
        if (orderStatus.isCancelled) {
            // Only revert if revertOnInvalid has been supplied as true.
            if (revertOnInvalid) {
                revert OrderIsCancelled(orderHash);
            }

            // Return false as the order status is invalid.
            return false;
        }

        // If the order is not entirely unused...
        if (orderStatus.numerator != 0) {
            // ensure the order has not been partially filled when not allowed.
            if (onlyAllowUnused) {
                // Always revert on partial fills when onlyAllowUnused is true.
                revert OrderPartiallyFilled(orderHash);
                // Otherwise, ensure that order has not been entirely filled.
            } else if (orderStatus.numerator >= orderStatus.denominator) {
                // Only revert if revertOnInvalid has been supplied as true.
                if (revertOnInvalid) {
                    revert OrderAlreadyFilled(orderHash);
                }

                // Return false as the order status is invalid.
                return false;
            }
        }

        // Return true as the order status is valid.
        valid = true;
    }
```

Recommendation

Differentiate fully filled orders from partially filled ones within `_verifyOrderStatus()`. Use a separate condition and revert message, such as `OrderAlreadyFilled(orderHash)`, to accurately reflect the state of the order.

9. Unchecked Arithmetic May Cause Overflow in Intermediate Calculations

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-09
Target	contracts/lib/AmountDeriver.sol
Function name	_locateCurrentAmount

Description

The function `_locateCurrentAmount` performs arithmetic involving multiplication and addition of potentially large values without explicit overflow checks. While Solidity 0.8+ includes built-in overflow protection, one part of the logic is wrapped in an `unchecked` block and others are not. This may cause transactions to revert unexpectedly due to overflow if the operands are large enough, potentially resulting in denial-of-service or inconsistent behavior.

```
function _locateCurrentAmount(
        uint256 startAmount,
        uint256 endAmount,
        uint256 elapsed,
        uint256 remaining,
        uint256 duration,
        bool roundUp
    ) internal pure returns (uint256) {
        // Only modify end amount if it doesn't already equal start amount.
        if (startAmount != endAmount) {
            // Leave extra amount to add for rounding at zero (i.e. round down).
            uint256 extraCeiling = 0;

            // If rounding up, set rounding factor to one less than denominator.
            if (roundUp) {
                // Skip underflow check: duration cannot be zero.
                unchecked {
                    extraCeiling = duration - 1;
                }
            }

            // Aggregate new amounts weighted by time with rounding factor
            // prettier-ignore
            uint256 totalBeforeDivision = (
                (startAmount * remaining) + (endAmount * elapsed) + extraCeiling
            );

            // Division performed with no zero check as duration cannot be zero.
            uint256 newAmount;
            assembly {
                newAmount := div(totalBeforeDivision, duration)
            }

            // Return the current amount (expressed as endAmount internally).
            return newAmount;
        }

        // Return the original amount (now expressed as endAmount internally).
        return endAmount;
    }
```

Recommendation

Wrap the full arithmetic expression involving `startAmount * remaining + endAmount * elapsed + extraCeiling` in an `unchecked` block if it is guaranteed not to overflow, or add validation checks to ensure the operands are within a safe range. Alternatively, use SafeMath or similar patterns to enforce bounds explicitly.

10. Hardcoded Calldata Offsets and Route Logic in Basic Order Processing

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-10
Target	contracts/lib/Consideration.sol
Function name	_validateAndFulfillBasicOrder

Description

The `_validateAndFulfillBasicOrder` function relies heavily on hardcoded calldata offsets and manually decoded logic in Yul/assembly. These hardcoded offsets correspond to values like `BasicOrder_considerationToken_cdPtr` and assume a fixed layout of `BasicOrderParameters`. This introduces maintainability and security risks. If the struct layout changes in the future, these offsets will become outdated, potentially leading to incorrect behavior or silent vulnerabilities. Additionally, the logic using bitwise operations to derive the route, orderType, and item types is opaque and error-prone.

```
function fulfillBasicOrder(BasicOrderParameters calldata parameters)
        external
        payable
        override
        returns (bool fulfilled)
    {
        // Validate and fulfill the basic order.
        fulfilled = _validateAndFulfillBasicOrder(parameters);
    }


function _validateAndFulfillBasicOrder(
        BasicOrderParameters calldata parameters
    ) internal returns (bool) {
        // Declare enums for order type & route to extract from basicOrderType.
        BasicOrderRouteType route;
        OrderType orderType;

        // Declare additional recipient item type to derive from the route type.
        ItemType additionalRecipientsItemType;
        // Utilize assembly to extract the order type and the basic order route.
        assembly {
            // Mask all but 2 least-significant bits to derive the order type.
            orderType := and(calldataload(BasicOrder_basicOrderType_cdPtr), 3)

            // Divide basicOrderType by four to derive the route.
            route := div(calldataload(BasicOrder_basicOrderType_cdPtr), 4)

            // If route > 1 additionalRecipient items are ERC20 (1) else Eth (0)
            additionalRecipientsItemType := gt(route, 1)
        }

        {
            // Declare temporary variable for enforcing payable status.
            bool correctPayableStatus;

            // Utilize assembly to compare the route to the callvalue.
            assembly {
                // route 0 and 1 are payable, otherwise route is not payable.
                correctPayableStatus := eq(
                    additionalRecipientsItemType,
                    iszero(callvalue())
                )
            }

            // Revert if msg.value has not been supplied as part of payable
            // routes or has been supplied as part of non-payable routes.
            if (!correctPayableStatus) {
                revert InvalidMsgValue(msg.value);
            }
        }

        // Declare more arguments that will be derived from route and calldata.
        address additionalRecipientsToken;
        ItemType receivedItemType;
        ItemType offeredItemType;

        // Utilize assembly to retrieve function arguments and cast types.
        assembly {
            // Determine if offered item type == additional recipient item type.
            let offerTypeIsAdditionalRecipientsType := gt(route, 3)

            // If route > 3 additionalRecipientsToken is at 0xc4 else 0x24.
            additionalRecipientsToken := calldataload(
                add(
                    BasicOrder_considerationToken_cdPtr,
                    mul(offerTypeIsAdditionalRecipientsType, FiveWords)
                )
            )

            // If route > 2, receivedItemType is route - 2. If route is 2, then
            // receivedItemType is ERC20 (1). Otherwise, it is Eth (0).
            receivedItemType := add(
                mul(sub(route, 2), gt(route, 2)),
                eq(route, 2)
            )

            // If route > 3, offeredItemType is ERC20 (1). If route is 2 or 3,
            // offeredItemType = route. If route is 0 or 1, it is route + 2.
            offeredItemType := sub(
                add(route, mul(iszero(additionalRecipientsItemType), 2)),
                mul(
                    offerTypeIsAdditionalRecipientsType,
                    add(receivedItemType, 1)
                )
            )
        }

        // Derive & validate order using parameters and update order status.
        _prepareBasicFulfillmentFromCalldata(
            parameters,
            orderType,
            receivedItemType,
            additionalRecipientsItemType,
            additionalRecipientsToken,
            offeredItemType
        );

        // Read offerer from calldata and place on the stack.
        address payable offerer = parameters.offerer;

        // Declare conduitKey argument used by transfer functions.
        bytes32 conduitKey;

        // Utilize assembly to derive conduit (if relevant) based on route.
        assembly {
            // use offerer conduit for routes 0-3, fulfiller conduit otherwise.
            conduitKey := calldataload(
                add(BasicOrder_offererConduit_cdPtr, mul(gt(route, 3), OneWord))
            )
        }

        // Transfer tokens based on the route.
        if (additionalRecipientsItemType == ItemType.NATIVE) {
            _transferIndividual721Or1155Item(
                offeredItemType,
                parameters.offerToken,
                offerer,
                msg.sender,
                parameters.offerIdentifier,
                parameters.offerAmount,
                conduitKey
            );

            // Transfer native to recipients, return excess to caller & wrap up.
            _transferEthAndFinalize(
                parameters.considerationAmount,
                offerer,
                parameters.additionalRecipients
            );
        } else {
            // Initialize an accumulator array. From this point forward, no new
            // memory regions can be safely allocated until the accumulator is
            // no longer being utilized, as the accumulator operates in an
            // open-ended fashion from this memory pointer; existing memory may
            // still be accessed and modified, however.
            bytes memory accumulator = new bytes(AccumulatorDisarmed);

            if (route == BasicOrderRouteType.ERC20_TO_ERC721) {
                // Transfer ERC721 to caller using offerer's conduit preference.
                _transferERC721(
                    parameters.offerToken,
                    offerer,
                    msg.sender,
                    parameters.offerIdentifier,
                    parameters.offerAmount,
                    conduitKey,
                    accumulator
                );

                // Transfer ERC20 tokens to all recipients and wrap up.
                _transferERC20AndFinalize(
                    msg.sender,
                    offerer,
                    parameters.considerationToken,
                    parameters.considerationAmount,
                    parameters.additionalRecipients,
                    false, // Send full amount indicated by consideration items.
                    accumulator
                );
            } else if (route == BasicOrderRouteType.ERC20_TO_ERC1155) {
                // Transfer ERC1155 to caller with offerer's conduit preference.
                _transferERC1155(
                    parameters.offerToken,
                    offerer,
                    msg.sender,
                    parameters.offerIdentifier,
                    parameters.offerAmount,
                    conduitKey,
                    accumulator
                );

                // Transfer ERC20 tokens to all recipients and wrap up.
                _transferERC20AndFinalize(
                    msg.sender,
                    offerer,
                    parameters.considerationToken,
                    parameters.considerationAmount,
                    parameters.additionalRecipients,
                    false, // Send full amount indicated by consideration items.
                    accumulator
                );
            } else if (route == BasicOrderRouteType.ERC721_TO_ERC20) {
                // Transfer ERC721 to offerer using caller's conduit preference.
                _transferERC721(
                    parameters.considerationToken,
                    msg.sender,
                    offerer,
                    parameters.considerationIdentifier,
                    parameters.considerationAmount,
                    conduitKey,
                    accumulator
                );

                // Transfer ERC20 tokens to all recipients and wrap up.
                _transferERC20AndFinalize(
                    offerer,
                    msg.sender,
                    parameters.offerToken,
                    parameters.offerAmount,
                    parameters.additionalRecipients,
                    true, // Reduce fulfiller amount sent by additional amounts.
                    accumulator
                );
            } else {
                // route == BasicOrderRouteType.ERC1155_TO_ERC20

                // Transfer ERC1155 to offerer with caller's conduit preference.
                _transferERC1155(
                    parameters.considerationToken,
                    msg.sender,
                    offerer,
                    parameters.considerationIdentifier,
                    parameters.considerationAmount,
                    conduitKey,
                    accumulator
                );

                // Transfer ERC20 tokens to all recipients and wrap up.
                _transferERC20AndFinalize(
                    offerer,
                    msg.sender,
                    parameters.offerToken,
                    parameters.offerAmount,
                    parameters.additionalRecipients,
                    true, // Reduce fulfiller amount sent by additional amounts.
                    accumulator
                );
            }

            // Trigger any remaining accumulated transfers via call to conduit.
            _triggerIfArmed(accumulator);
        }

        return true;
    }
```

Recommendation

Replace hardcoded offsets and complex assembly-based decoding with Solidity-native decoding whenever possible. If performance demands manual decoding, define constants via compile-time logic or tightly document the offset structure and enforce offset consistency via unit tests. Ensure that any layout changes trigger failing tests.

11. Missing Zero-Division Check in _getFraction May Cause Unexpected Reverts

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-11
Target	contracts/lib/AmountDeriver.sol
Function name	_getFraction

Description

The _getFraction function in AmountDeriver.sol assumes that the denominator is non-zero, but this is never explicitly checked. While mulmod with a zero denominator does not revert and returns 0, the subsequent division will revert unexpectedly if denominator is zero. This leads to inconsistent error handling as the revert will not use the intended custom InexactFraction error, and may be difficult to trace or debug in production.

```
function _applyFraction(
        uint256 startAmount,
        uint256 endAmount,
        uint256 numerator,
        uint256 denominator,
        uint256 elapsed,
        uint256 remaining,
        uint256 duration,
        bool roundUp
    ) internal pure returns (uint256 amount) {
        // If start amount equals end amount, apply fraction to end amount.
        if (startAmount == endAmount) {
            // Apply fraction to end amount.
            amount = _getFraction(numerator, denominator, endAmount);
        } else {
            // Otherwise, apply fraction to both and extrapolate final amount.
            amount = _locateCurrentAmount(
                _getFraction(numerator, denominator, startAmount),
                _getFraction(numerator, denominator, endAmount),
                elapsed,
                remaining,
                duration,
                roundUp
            );
        }
    }
}

function _getFraction(
        uint256 numerator,
        uint256 denominator,
        uint256 value
    ) internal pure returns (uint256 newValue) {
        // Return value early in cases where the fraction resolves to 1.
        if (numerator == denominator) {
            return value;
        }

        // Ensure fraction can be applied to the value with no remainder. Note
        // that the denominator cannot be zero.
        bool exact;
        assembly {
            // Ensure new value contains no remainder via mulmod operator.
            // Credit to @hrkrshnn + @axic for proposing this optimal solution.
            exact := iszero(mulmod(value, numerator, denominator))
        }

        // Ensure that division gave a final result with no remainder.
        if (!exact) {
            revert InexactFraction();
        }

        // Multiply the numerator by the value and ensure no overflow occurs.
        uint256 valueTimesNumerator = value * numerator;

        // Divide and check for remainder. Note that denominator cannot be zero.
        assembly {
            // Perform division without zero check.
            newValue := div(valueTimesNumerator, denominator)
        }
    }
```

Recommendation

Add an explicit check at the start of the _getFraction function to ensure denominator is non-zero. This can be done using a require statement or a custom error to maintain consistency with the contract's error handling approach.

12. Manual Gas Estimation in `_revertWithReasonIfOneIsReturned` Risks EVM Divergence

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-12
Target	contracts/lib/LowLevelHelpers.sol
Function name	_revertWithReasonIfOneIsReturned

Description

The `_revertWithReasonIfOneIsReturned` function in `LowLevelHelpers.sol` implements custom gas and memory cost estimation logic to determine whether to bubble up revert data. This manual calculation attempts to replicate the EVM's memory expansion behavior and gas usage, which may diverge from actual behavior in different compiler versions or future EVM upgrades. If the estimation is inaccurate, it may cause the function to incorrectly assume sufficient gas is available, resulting in a full transaction revert.

```
function _revertWithReasonIfOneIsReturned() internal view {
        assembly {
            if returndatasize() {
                let returnDataWords := div(returndatasize(), OneWord)
                let msizeWords := div(mload(FreeMemoryPointerSlot), OneWord)
                let cost := mul(CostPerWord, returnDataWords)
                if gt(returnDataWords, msizeWords) {
                    cost := add(
                        cost,
                        add(
                            mul(sub(returnDataWords, msizeWords), CostPerWord),
                            div(
                                sub(
                                    mul(returnDataWords, returnDataWords),
                                    mul(msizeWords, msizeWords)
                                ),
                                MemoryExpansionCoefficient
                            )
                        )
                    )
                }
                if lt(add(cost, ExtraGasBuffer), gas()) {
                    returndatacopy(0, 0, returndatasize())
                    revert(0, returndatasize())
                }
            }
        }
    }
```

Recommendation

Avoid manually replicating EVM internals for gas and memory cost estimation where possible. Instead, consider using more conservative checks or simplifying the bubbling logic. Clearly document any assumptions about the EVM gas cost model if the logic is retained.

13. Unchecked Overflow in Signature Offset Calculation in `_assertValidBasicOrderParameterOffsets`

Severity	Low
Finding ID	BH-2022-05-opensea-seaport-13
Target	contracts/lib/Assertions.sol
Function name	_assertValidBasicOrderParameterOffsets

Description

The `_assertValidBasicOrderParameterOffsets` function performs offset checks on calldata fields using inline assembly. However, it does not validate whether the multiplication `len * 0x40` overflows when computing the expected signature offset. An attacker can craft calldata with a manipulated `additionalRecipients.length` field that overflows the multiplication, allowing bypass of the offset check and potentially leading to incorrect memory access or security issues.

```
function _assertValidBasicOrderParameterOffsets() internal pure {
        // Declare a boolean designating basic order parameter offset validity.
        bool validOffsets;

        // Utilize assembly in order to read offset data directly from calldata.
        assembly {
            /*
             * Checks:
             * 1. Order parameters struct offset == 0x20
             * 2. Additional recipients arr offset == 0x240
             * 3. Signature offset == 0x260 + (recipients.length * 0x40)
             */
            validOffsets := and(
                // Order parameters at calldata 0x04 must have offset of 0x20.
                eq(
                    calldataload(BasicOrder_parameters_cdPtr),
                    BasicOrder_parameters_ptr
                ),
                // Additional recipients at cd 0x224 must have offset of 0x240.
                eq(
                    calldataload(BasicOrder_additionalRecipients_head_cdPtr),
                    BasicOrder_additionalRecipients_head_ptr
                )
            )
            validOffsets := and(
                validOffsets,
                eq(
                    // Load signature offset from calldata 0x244.
                    calldataload(BasicOrder_signature_cdPtr),
                    // Derive expected offset as start of recipients + len * 64.
                    add(
                        BasicOrder_signature_ptr,
                        mul(
                            // Additional recipients length at calldata 0x264.
                            calldataload(
                                BasicOrder_additionalRecipients_length_cdPtr
                            ),
                            // Each additional recipient has a length of 0x40.
                            AdditionalRecipients_size
                        )
                    )
                )
            )
        }

        // Revert with an error if basic order parameter offsets are invalid.
        if (!validOffsets) {
            revert InvalidBasicOrderParameterEncoding();
        }
    }
```

Recommendation

Add a bounds check to ensure that `len * 0x40` does not overflow. This can be done by asserting that `len` is below a safe threshold (e.g., `len < 2**252`) before performing the multiplication.