OpenSea Seaport contest Findings & Analysis Report

2022-08-30

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the OpenSea Seaport smart contract system written in Solidity. The audit contest took place between May 20—June 3 2022.

Wardens

65 Wardens contributed reports to the OpenSea Seaport contest:

1. Spearbit
2. Saw-mon_and_Natalie
3. cmichel
4. Oxsanson
5. frangio
6. broccoli (shw and jonah1005)
7. OriDabush
8. hyh

9. ming
10. Yarpo
11. lllllll
12. shung
13. Chom
14. Dravee
15. zkhorse (karmacoma and horsefacts)
16. sces60107
17. peritoflores
18. hickuphh3
19. hack3r-0m
20. ilan
21. cccz
22. csanuragjain
23. rfa
24. oyc_109
25. twojoy
26. foobar
27. mayo
28. scaraven
29. kebabsec (okkothejawa and FlameHorizon)
30. sorrynotsorry
31. zzzitron
32. tintin
33. hubble (ksk2345 and shri4net)
34. 0x1f8b
35. NoamYakov
36. djxploit
37. 0xalpharush

38. Czar102
39. 0x29A (0x4non and rotcivegaf)
40. sirhashalot
41. gzeon
42. MaratCerby
43. zer0dot
44. defsec
45. Hawkeye (0xwags and 0xmint)
46. ignacio
47. joestakey
48. MiloTruck
49. sashik_eth
50. kaden
51. sach1r0
52. TomJ
53. ellahi
54. TerrierLover
55. asutorufos
56. delfin454000
57. hake
58. RoiEvenHaim
59. Tadashi

This contest was judged by 0xleastwood and HardlyDifficult. Additional judging assistance provided by Alex the Entreprenerd for reports detailing low risk and non-critical issues.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 2 received a risk rating in the category of MEDIUM severity. Per OpenSea, all of these HIGH and MEDIUM severity findings have been addressed via Seaport 1.1. (see specific mitigations linked on each finding in the sections below)

Additionally, C4 analysis included 29 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 45 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

Seaport 1.0 was the focus of this audit contest. The code under review can be found within the C4 OpenSea Seaport contest repository, and is composed of 44 smart contracts written in the Solidity programming language and includes 9,771 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] Truncation in `OrderValidator` can lead to resetting the fill and selling more tokens

Submitted by Spearbit, also found by 0xsanson, broccoli, cmichel, hyh, ming, OriDabush, Saw-mon_and_Natalie, and Yarpo

OrderValidator.sol#L228
OrderValidator.sol#L231
OrderValidator.sol#L237
OrderValidator.sol#L238

A partial order's fractions (`numerator` and `denominator`) can be reset to `0` due to a truncation. This can be used to craft malicious orders:

1. Consider user Alice, who has 100 ERC1155 tokens, who approved all of their tokens to the `marketplaceContract` .
2. Alice places a `PARTIAL_OPEN` order with 10 ERC1155 tokens and consideration of ETH.
3. Malory tries to fill the order in the following way:
    1. Malory tries to fill 50% of the order, but instead of providing the fraction `1 / 2` , Bob provides `2**118 / 2**119` . This sets the `totalFilled` to `2**118` and `totalSize` to `2**119` .
    2. Malory tries to fill 10% of the order, by providing `1 / 10` . The computation `2**118 / 2**119 + 1 / 10` is done by "cross multiplying" the denominators, leading to the acutal fraction being `numerator = (2**118 * 10 + 2**119)` and `denominator = 2**119 * 10` .
    3. Because of the `uint120` truncation in OrderValidator.sol#L228-L248, the `numerator` and `denominator` are truncated to `0` and `0` respectively.
    4. Bob can now continue filling the order and draining any approved (1000 tokens in total) of the above ERC1155 tokens, for the same consideration amount!

Proof of Concept

View full POC.

The following change would make the above POC fail:

```
modified   contracts/lib/OrderValidator.sol
@@ -225,6 +225,8 @@ contract OrderValidator is Executor, ZoneInt
                // Update order status and fill amount, packing
                _orderStatus[orderHash].isValidated = true;
                _orderStatus[orderHash].isCancelled = false;
+               require(filledNumerator + numerator <= type(uir
+               require(denominator <= type(uint120).max, "over
                _orderStatus[orderHash].numerator = uint120(
                    filledNumerator + numerator
                );
@@ -234,6 +236,8 @@ contract OrderValidator is Executor, ZoneInt
            // Update order status and fill amount, packing str
            _orderStatus[orderHash].isValidated = true;
            _orderStatus[orderHash].isCancelled = false;
+           require(numerator <= type(uint120).max, "overflow")
+           require(denominator <= type(uint120).max, "overflow
            _orderStatus[orderHash].numerator = uint120(numerat
            _orderStatus[orderHash].denominator = uint120(denom
        }
```

Recommended Mitigation Steps

A basic fix for this would involve adding the above checks for overflow / truncation and reverting in that case. However, we think the mechanism is still flawed in some respects and requires more changes to fully fix it. See a related issue: "A malicious filler can fill a partial order in such a way that the rest cannot be filled by anyone" that points out a related but a more fundamental issue with the mechanism.

0age (OpenSea) confirmed

0xleastwood (judge) commented:

> I've identified that this issue and all of its duplicates clearly outline how an attacker might overflow an order to continually fulfill an order at the same market price.

> An instance where this issue might cause issues is during a restricted token sale. A relevant scenario is detailed as follows:

- A new token is created and the owner wishes to sell 50% of the token supply to the public.
- Because of an edge case in `OrderValidator`, the order fulfillment can be reset to allow the public to more than 50% of the total token supply.
- As a result, allocations intended to be distributed to investors and the team, will no longer be available.
- It is important to note, that additional tokens will be sold at the intended market price listed by the original order.

> For these reasons, I believe this issue to be of high severity because it breaks certain trust assumptions made by the protocol and its userbase. By intentionally forcing a user to sell additional tokens, you are effectively altering the allocation of their wallet holdings, potentially leading to further funds loss as they may incur slippage when they have to sell these tokens back.

> A great finding from all involved!

0age (OpenSea) resolved:

> PR: ProjectOpenSea/seaport#319

[H-02] `_aggregateValidFulfillmentOfferItems()` can be tricked to accept invalid inputs

Submitted by Spearbit, also found by Saw-mon_and_Natalie

FulfillmentApplier.sol#L406

The `_aggregateValidFulfillmentOfferItems()` function aims to revert on orders with zero value or where a total consideration amount overflows. Internally this is accomplished by having a temporary variable `errorBuffer`, accumulating issues found, and only reverting once all the items are processed in case there was a problem found. This code is optimistic for valid inputs.

Note: there is a similar issue in `_aggregateValidFulfillmentConsiderationItems()`, which is reported separately.

The problem lies in how this `errorBuffer` is updated:

```
// Update error buffer (1 = zero amount, 2 = ove
errorBuffer := or(
  errorBuffer,
  or(
    shl(1, lt(newAmount, amount)),
    iszero(mload(amountPtr))
  )
)
```

The final error handling code:

```
// Determine if an error code is contained in the er
switch errorBuffer
case 1 {
    // Store the MissingItemAmount error signature.
    mstore(0, MissingItemAmount_error_signature)

    // Return, supplying MissingItemAmount signature
    revert(0, MissingItemAmount_error_len)
}
case 2 {
    // If the sum overflowed, panic.
    throwOverflow()
}
```

While the expected value is `0` (success), `1` or `2` (failure), it is possible to set it to `3`, which is unhandled and considered as a "success". This can be easily accomplished by having both an overflowing item and a zero item in the order list.

This validation error could lead to fulfilling an order with a consideration (potentially ~0) lower than expected.

Proof of Concept

Craft an offer containing two errors (e.g. with zero amount and overflow).
Call `matchOrders()`. Via calls to `_matchAdvancedOrders()`, `_fulfillAdvancedOrders()`, `_applyFulfillment()`, `_aggregateValidFulfillmentOfferItems()` will be called.
The `errorBuffer` will get a value of 3 (the `or` of 1 and 2).
As the value of 3 is not detected, no error will be thrown and the order will be executed, including the mal formed values.

Recommended Mitigation Steps

1. Change the check on FulfillmentApplier.sol#L465 to consider `case 3`.
2. Potential option: Introduce an early abort in case `errorBuffer != 0` on FulfillmentApplier.sol#L338

0age (OpenSea) confirmed

HardlyDifficult (judge) decreased severity to Medium

cmichel (warden) commented:

> This validation error could lead to fulfilling an order with a consideration (potentially ~0) lower than expected.

> That's correct, you can use this to fulfill an order essentially for free, that's why I'd consider this high severity. They could have done a better job demonstrating it with a POC test case but this sentence imo shows that they were aware of the impact.

> See this test case showing how to buy an NFT for 1 DAI instead of 1000 DAI.

0age (OpenSea) disagreed with Medium severity:

> This is the highest-severity finding. If it were me, I'd switch this to high.

HardlyDifficult (judge) increased severity to High

0xleastwood (judge) commented:

> After further consideration and discussion with @HardlyDifficult, we agree with @cmichel that this should be of high severity. As the protocol allows for invalid orders to be created, users aware of this vulnerability will be able to fulfill an order at a considerable discount. This fits the criteria of a high severity issue as it directly leads to lost funds.

0age (OpenSea) resolved:

> PR: ProjectOpenSea/seaport#320

Medium Risk Findings (2)

[M-01] Merkle Tree criteria can be resolved by wrong tokenIDs

Submitted by cmichel, also found by frangio and Spearbit

CriteriaResolution.sol#L157

The protocol allows specifying several tokenIds to accept for a single offer.
A merkle tree is created out of these tokenIds and the root is stored as the `identifierOrCriteria` for the item.
The fulfiller then submits the actual tokenId and a proof that this tokenId is part of the merkle tree.

There are no real verifications on the merkle proof that the supplied tokenId is indeed **a leaf of the merkle tree.**
It's possible to submit an intermediate hash of the merkle tree as the tokenId and trade this NFT instead of one of the requested ones.

This leads to losses for the offerer as they receive a tokenId that they did not specify in the criteria.
Usually, this criteria functionality is used to specify tokenIds with certain traits that are highly valuable. The offerer receives a low-value token that does not have these traits.

Example

Alice wants to buy either NFT with tokenId 1 or tokenId 2.
She creates a merkle tree of it and the root is `hash(1||2) = 0xe90b7bceb6e7df5418fb78d8ee546e97c83a08bbccc01a0644d599ccd2a7c2e0`.
She creates an offer for this criteria.
An attacker can now acquire the NFT with tokenId `0xe90b7bceb6e7df5418fb78d8ee546e97c83a08bbccc01a0644d599ccd2a7c2e0` (or, generally, any other intermediate hash value) and fulfill the trade.

> One might argue that this attack is not feasible because the provided hash is random and tokenIds are generally a counter. However, this is not required in the standard.

> "While some ERC-721 smart contracts may find it convenient to start with ID 0 and simply increment by one for each new NFT, callers SHALL NOT assume that ID numbers have any specific pattern to them, and MUST treat the ID as a 'black box'." EIP721

> Neither do the standard OpenZeppelin/Solmate implementations use a counter. They only provide internal `_mint(address to, uint256 id)` functions that allow specifying an arbitrary `id`. NFT contracts could let the user choose the token ID to mint, especially contracts that do not have any linked off-chain metadata like Uniswap LP positions.
> Therefore, ERC721-compliant token contracts are vulnerable to this attack.

Proof of Concept

Here's a `forge` test (gist) that shows the issue for the situation mentioned in *Example*.

```
contract BugMerkleTree is BaseOrderTest {
    struct Context {
        ConsiderationInterface consideration;
        bytes32 tokenCriteria;
        uint256 paymentAmount;
        address zone;
        bytes32 zoneHash;
        uint256 salt;
    }
```

```
function hashHashes(bytes32 hash1, bytes32 hash2)
    internal
    returns (bytes32)
{
    // see MerkleProof.verify
    bytes memory encoding;
    if (hash1 <= hash2) {
        encoding = abi.encodePacked(hash1, hash2);
    } else {
        encoding = abi.encodePacked(hash2, hash1);
    }
    return keccak256(encoding);
}

function testMerkleTreeBug() public resetTokenBalancesBetwee
    // Alice wants to buy NFT ID 1 or 2 for token1. compute
    bytes32 leafLeft = bytes32(uint256(1));
    bytes32 leafRight = bytes32(uint256(2));
    bytes32 merkleRoot = hashHashes(leafLeft, leafRight);
    console.logBytes32(merkleRoot);

    Context memory context = Context(
        consideration,
        merkleRoot, /* tokenCriteria */
        1e18, /* paymentAmount */
        address(0), /* zone */
        bytes32(0), /* zoneHash */
        uint256(0) /* salt */
    );
    bytes32 conduitKey = bytes32(0);

    token1.mint(address(alice), context.paymentAmount);
    // @audit assume there's a token where anyone can acquir
    // we acquire the merkle root ID
    test721_1.mint(address(this), uint256(merkleRoot));

    _configureERC20OfferItem(
        // start, end
        context.paymentAmount, context.paymentAmount
    );
    _configureConsiderationItem(
        ItemType.ERC721_WITH_CRITERIA,
        address(test721_1),
        // @audit set merkle root for NFTs we want to accept
        uint256(context.tokenCriteria), /* identifierOrCrite
```

```
            1,
            1,
            alice
        );

        OrderParameters memory orderParameters = OrderParameters
            address(alice),
            context.zone,
            offerItems,
            considerationItems,
            OrderType.FULL_OPEN,
            block.timestamp,
            block.timestamp + 1000,
            context.zoneHash,
            context.salt,
            conduitKey,
            considerationItems.length
        );

        OrderComponents memory orderComponents = getOrderCompone
            orderParameters,
            context.consideration.getNonce(alice)
        );
        bytes32 orderHash = context.consideration.getOrderHash(c
        bytes memory signature = signOrder(
            context.consideration,
            alicePk,
            orderHash
        );

        delete offerItems;
        delete considerationItems;

        /*************** ATTACK STARTS HERE ***************/
        AdvancedOrder memory advancedOrder = AdvancedOrder(
            orderParameters,
            1, /* numerator */
            1, /* denominator */
            signature,
            ""
        );

        // resolve the merkle root token ID itself
        CriteriaResolver[] memory cr = new CriteriaResolver[](1)
        bytes32[] memory proof = new bytes32[](0);
        cr[0] = CriteriaResolver(
```

```
                0, // uint256 orderIndex;
                Side.CONSIDERATION, // Side side;
                0, // uint256 index; (item)
                uint256(merkleRoot), // uint256 identifier;
                proof // bytes32[] criteriaProof;
            );

            uint256 profit = token1.balanceOf(address(this));
            context.consideration.fulfillAdvancedOrder{
                value: context.paymentAmount
            }(advancedOrder, cr, bytes32(0));
            profit = token1.balanceOf(address(this)) - profit;

            // @audit could fulfill order without owning NFT 1 or 2
            assertEq(profit, context.paymentAmount);
        }
    }
```

Recommended Mitigation Steps

Usually, this is fixed by using a type-byte that indicates if one is computing the hash for a *leaf* or not.

An elegant fix here is to simply use hashes of the tokenIds as the leaves - instead of the tokenIds themselves. (Note that this is the natural way to compute merkle trees if the data size is not already the hash size.)

Then compute the leaf hash in the contract from the provided tokenId:

```
    function _verifyProof(
        uint256 leaf,
        uint256 root,
        bytes32[] memory proof
    ) internal pure {
        bool isValid;

-       assembly {
-           let computedHash := leaf
+      bytes32 computedHash = keccak256(abi.encodePacked(leaf))
      ...
```

There can't be a collision between a leaf hash and an intermediate hash anymore as the former is the result of hashing 32 bytes, while the latter are the results of hashing

64 bytes.

Note that this requires off-chain changes to how the merkle tree is generated. (Leaves must be hashed first.)

0age (OpenSea) confirmed, but disagreed with severity

HardlyDifficult (judge) decreased severity to Medium

0xleastwood (judge) commented:

> The attack outlined by the warden showcases how an intermediate node of a proof can be used as leaves, potentially allowing the attacker to resolve the merkle tree to a different `tokenId`. I think in the majority of cases, this will not allow users to trade on invalid `tokenIds`, however, considering the `ERC721` specification does not enforce a standard for how NFTs are represented using `tokenIds`, the issue has some legitimacy. Because of this, I believe `medium` severity to be justified.

0age (OpenSea) resolved:

> PR: ProjectOpenSea/seaport#316

[M-02] Wrong items length assertion in basic order

Submitted by 0xsanson, also found by cmichel

BasicOrderFulfiller.sol#L346-L349

When fulfilling a basic order we need to assert that the parameter `totalOriginalAdditionalRecipients` is less or equal than the length of `additionalRecipients` written in calldata.
However in `_prepareBasicFulfillmentFromCalldata` this assertion is incorrect (L346):

```
// Ensure supplied consideration array length is not les
_assertConsiderationLengthIsNotLessThanOriginalConsidera
```

```
            parameters.additionalRecipients.length + 1,
            parameters.totalOriginalAdditionalRecipients
        );
```

The way the function is written (L75), it accepts also a length smaller than the original by 1 (basically there shouldn't be a `+ 1` in the first argument).

Interestingly enough, in the case `additionalRecipients.length < totalOriginalAdditionalRecipients`, the inline-assembly for-loop at (L506) will read consideration items out-of-bounds.
This can be a vector of exploits, as illustrated below.

Proof of Concept

Alice makes the following offer: a basic order, with two `considerationItem`s. The second item has the following data:

```
consideration[1] = {
        itemType: ...,
        token: ...,
        identifierOrCriteria: ...,
        startAmount: X,
        endAmount: X,
        recipient: Y,
}
```

The only quantities we need to track are the amounts `X` and recipient `Y`.

When fulfilling the order normally, the fulfiller will spend `X` tokens sending them to `Y`. It's possible however to exploit the previous bug in a way that the fulfiller won't need to make this transfer.

To do this, the fulfiller needs to craft the following calldata:

calldata pointer	correct calldata	exploit calldata
...	...	...
0x204	1 (tot original)	1 (tot original)

calldata pointer	correct calldata	exploit calldata
0x224	0x240 (head addRec)	0x240 (head addRec)
0x244	0x2a0 (head sign)	0x260 (head sign)
0x264	1 (length addRec)	0 (length addRec)
0x284	X (amount)	X (length sign)
0x2a4	Y (recipient)	Y (sign body)
0x2c4	0x40 (length sign)	0x00 (sign body)
0x2e4	[correct Alice sign]	...
0x304	[correct Alice sign]	...

Basically writing `additionalRecipients = []` and making the signature length = `X`, with `Y` being the first 32 bytes. Of course this signature will be invalid; however it doesn't matter since the exploiter can call `validate` with the correct signature beforehand.

The transaction trace will look like this:

- the assertion `_assertConsiderationLengthIsNotLessThanOriginalConsiderationLength` passes;
- the `orderHash` calculated is the correct one, since the for-loop over original consideration items picks up calldata at pointers {0x284, 0x2a4} **(L513)**;
- the order was already validated beforehand, so the signature isn't read;
- at the end, during the tokens transfers, only `offer` and `consideration[0]` are transferred, since the code looks at `additionalRecipients` which is empty.

Conclusion:

Every Order that is "basic" and has two or more consideration items can be fulfilled in a way to not trade the *last* consideration item in the list. The fulfiller spends less then normally, and a recipient doesn't get his due.

There's also an extra requirement which is stricter: this last item's `startAmount` (= endAmount) needs to be smallish (< 1e6). This is because this number becomes the signature bytes length, and we need to fill the calldata with extra zeroes to complete it. Realistically then the exploit will work only if the item is a ERC20 will low decimals.

I've made a hardhat test that exemplifies the exploit. **(Link to gist)**

Recommended Mitigation Steps

Remove the `+1` at L347.

0age (OpenSea) confirmed, but disagreed with severity and commented:

> Valid finding on the off-by-one error, this was already reported to us outside of c4 and we're going to fix — will mention that it's very difficult to find / craft exploitable payloads though, so severity should be lower.

HardlyDifficult (judge) decreased severity to Medium

0xleastwood (judge) commented:

> While the issue outlines an exploit whereby an attacker can fulfill an order without paying the entire consideration amount, it does require a set of requirements, namely:
>
> - The item is an ERC20 with low decimals.
> - The order has `considerationItems > 1`.
>
> Maximum extractable value for the most prevalent ERC20 token with low decimals, `WBTC`. This token uses 8 decimals and currently we know that calldata uses 16 gas for each byte used. Based on a block gas limit of `30,000,000`, we can deduce that the calldata length has an upper bound of `1.875` MB. Based on this, the maximum extractable value would be `(1,875,000 / 1e8) * $20,000 USD = $375 USD`, assuming the price for each `WBTC` is $20,000 USD.
>
> Relevant EIP detailing this is found at https://eips.ethereum.org/EIPS/eip-4488.

> It is also important to note, that by utilising the entire available block space on Ethereum, it is very likely that the cost of the transaction will far exceed the amount received in the attack.

> The attack does in fact leak value by allowing orders to be fulfilled at a slight discount. However, because this only affects very specific order types, I believe `medium` severity to be justified.

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> This was one of the most interesting issues I've read, kudos to those who found it!

0age (OpenSea) resolved:

> PR: ProjectOpenSea/seaport#317

Low Risk and Non-Critical Issues

For this contest, 29 reports were submitted by wardens detailing low risk and non-critical issues. Many of these reports were integrated into Seaport 1.1, often by the warden themselves; see **this PR** from OpenSea for the full set of changes.

The **report highlighted below** by team **Spearbit** received the top score from the judge.

The following wardens also submitted reports: **Saw-mon_and_Natalie**, **cmichel**, **IIIIIII**, **broccoli**, **Chom**, **sces60107**, **zkhorse**, **shung**, **hack3r-0m**, **peritoflores**, **OriDabush**, **hyh**, **scaraven**, **hickuphh3**, **ilan**, **cccz**, **0xsanson**, **csanuragjain**, **kebabsec**, **sorrynotsorry**, **zzzitron**, **oyc_109**, **twojoy**, **tintin**, **rfa**, **foobar**, **hubble**, *and* **mayo**.

Table of Contents

order

- [03] `_performERC1155BatchTransfers` consumes all gas on invalid input
- [04] Hardcoded values in `_validateAndFulfillBasicOrder()`
- [05] Helpers should have their expectations explained in NatSpec/comments
- [06] `ConduitTransfer` identifier field can be dirty and unused
- [07] Inconsistent way to return values in `_verifyTime()` and `_verifyOrderStatus`
- [08] `_callConduitUsingOffsets` depends on compiler behaviour for inter-assembly-block cleanup
- [09] `Assertions.sol` is missing an assertion about bounds for array length
- [10] LowLevelHelpers enforce a stricter ABI standard for `returndatasize`
- [11] `_assertValidSignature` allows malleable `secp256k1` signatures
- [12] The memory cost calculation logic `_revertWithReasonIfOneIsReturned` is duplicated in multiple places
- [13] Accumulator code depends on memory usage
- [14] Doing a basic order twice gives a misleading error message
- [15] Orders with moving price and `endTime = type(uint).max` can never be fulfilled
- [16] `_prepareBasicFulfillmentFromCalldata` overwrites memory extensively
- [17] Parameters passed to `fulfillBasicOrder()` insufficiently checked
- [18] `startAmount` and `endAmount` being different would severely limit partial orders
- [19] SafeTransferFrom: `transferFrom` to precompiles may succeed, a deviation from OZ's implementation
- [20] The gas computation for memory expansion rounds down instead of rounding up
- [21] `TokenTransferrer._performERC1155BatchTransfers` leaves corrupted memory
- [22] `_triggerIfArmed()` done outside of reentrancy guards

- [23] Deviations between Solidity compiler's checks and seaport's checks in `validateOrderParameters`

[01] The function `_name()` returns dirty data

Context: Seaport.sol#L41, ConsiderationBase.sol#L100

The function `name()` of Seaport.sol returns dirty data.

This may create issues with frontends that expect clean data. In fact, Etherscan is having trouble decoding it!

https://etherscan.io/address/0x00000000006cee72100d161c57ada5bb2be1ca79#readContract and click name! There is a junk character at the end.

This also could have negative impact on composability.

The external function `name()` gets its value from `_name()` in contract Seaport.

```
contract ReferenceConsideration is ConsiderationInterface, Refer
    ...
    function name() external pure override returns (string memor
        contractName = _name();
    }
    ...
}


contract Seaport is Consideration {
    ...
    function _name() internal pure override returns (string memo
        // Return the name of the contract.
        assembly {
            mstore(0, 0x20)
            mstore(0x27, 0x07536561706f7274)
            return(0, 0x60)
        }
    }
}
```

Function `_name()` is supposed to return “Seaport”. The ABI encoded data has offset as the first 32 bytes (`0x20` is the offset). The offset has info length `7` followed by “Seaport” (`0x536561706f7274` but padded with zeros on the right).

Properly encoded data would look like this:

```
0000000000000000000000000000000000000000000000000000000020
0000000000000000000000000000000000000000000000000000000007
536561706f7274000000000000000000000000000000000000000000
```

The final `mstore(0x27, ...)` only writes to memory regions `[0x29, 0x49)`. The remainder will likely contain junk. You can expect the actual data to be:

```
0000000000000000000000000000000000000000000000000000000020
0000000000000000000000000000000000000000000000000000000007
536561706f72740????????????????????????????????????????
```

Because `0x40` points to the free memory pointer, you can expect the value `mload(0x40)` to be a relatively small number. So only the last few least significant bits would be set, with the rest to be `0` in usual cases.

Proof of Concept

```
modified   test/foundry/FulfillBasicOrderTest.sol
@@ -32,6 +32,25 @@ contract FulfillBasicOrderTest is BaseOrderTe
         uint128 tokenAmount;
     }

+    function testName() public {
+        string memory name = consideration.name();
+        uint rds;
+        uint a;
+        uint b;
+        bytes32 c;
+        assembly {
+            rds := returndatasize()
+            returndatacopy(mload(0x40), 0, returndatasize())
```

```
+            a := mload(mload(0x40))
+            b := mload(add(mload(0x40), 32))
+            c := mload(add(mload(0x40), 64))
+        }
+        emit log_named_uint("returndatasize: ", rds);
+        emit log_named_uint("offset: ", a);
+        emit log_named_uint("length: ", b);
+        emit log_named_bytes32("data: ", c);
+    }
+
```

returns:

```
  returndatasize: : 96
  offset: : 32
  length: : 13
  data: : 0x436f6e7369646572617469f6e0000000000000000000000000(
```

The final `80` character is the initial value of the free memory pointer!

Recommended Mitigation Steps

Clean the last word properly. This requires an additional `mstore`.

HEVM confirmation

hevm confirms this.
File 1:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.13;

contract Seaport {
        function f() public pure returns (string memory) {
                return _name();
        }

    function _name() internal pure returns (string memory) {
        // Return the name of the contract.
        assembly {
            mstore(0, 0x20)
```

```
                mstore(0x27, 0x07536561706f7274)
                return(0, 0x60)
            }
        }
    }
```

File 2:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.13;

contract Seaport {
        function f() public pure returns (string memory) {
                return _nameString();
        }

    function _nameString() internal pure returns (string memory)
        // Return the name of the contract.
        return "Seaport";
    }
}
```

Run:

```
hevm equivalence --code-a $(cat seaport1.bin) --code-b $(cat sea
Not equal!
Counterexample:
Calldata:
0x26121ff0000000000000000000000000000000
Caller:
0x0000000000000000000000000000000000000000
Callvalue:
0
```

Adding `mstore(0x49, 0x00)` before the return in the first version makes it work:

```
hevm equivalence --code-a $(cat seaport1.bin) --code-b $(cat sea
Explored: 4 execution paths of A and: 4 paths of B.
No discrepancies found.
```

[02] `_revertWithReasonIfOneIsReturned`, `_doesNotMatchMagic` (and `_assertIsValidOrderStaticcallSuccess`) have a fragile dependency on call order

These helper functions rely on the undisturbed contents returned by `returndatasize` / `returndatacopy`. Should the call sites undergo some changes, they may not function as intended. They are used in numerous functions.

This is further complicated in `_assertIsValidOrderStaticcallSuccess` which is another layer hiding this assumption.

Proof of Concept

Context: LowLevelHelpers.sol#L46, LowLevelHelpers.sol#L103, ZoneInteraction.sol#L157

Issues can arise if:

1. The order of these helpers change and some other call is performed
2. These helpers become non-internal library functions (because then a call is performed and in the new context the buffer is empty)

An example use case is below:

```
        assembly {
            // Transfer the ETH and store if it succeeded or not
            success := call(gas(), to, amount, 0, 0, 0, 0)
        }

        // <-- Do something here to disturb the returndata buffer

        // If the call fails...
        if (!success) {
            // Revert and pass the revert reason along if one wa
            _revertWithReasonIfOneIsReturned();

            // Otherwise, revert with a generic error message.
            revert EtherTransferGenericFailure(to, amount);
```

```
        }
```

Simple way to make this fail:

```
-     function _revertWithReasonIfOneIsReturned() internal view {
+     function _revertWithReasonIfOneIsReturned() view {
```

Recommended Mitigation Steps

Document these assumptions as a warning and carefully test these cases.

[03] `_performERC1155BatchTransfers` consumes all gas on invalid input

The `_performERC1155BatchTransfers` function manually ABI decodes `ConduitBatch1155Transfer[]`. While doing so it will read field, without any validation, to determine the length of expected data and subsequently execute `calldatacopy` with potentially unbounded copying.

This may be used as a griefing attack, and such attacks seem to be attempted to be avoided by the project, as evidenced by the logic in `_revertWithReasonIfOneIsReturned`.

This function is only used in the conduit and is exposed under the `executeBatch1155` external function. The risk depends on how this will be used in the future.

Proof of Concept

Context: TokenTransferrer.sol#L524, TokenTransferrer.sol#L530, TokenTransferrer.sol#L557

Recommended Mitigation Steps

Validate lengths against `calldatasize()`.

[04] Hardcoded values in `_validateAndFulfillBasicOrder()`

The distance between fields in structs is hardcoded. This could lead to mistakes with future maintenance of the code. This distance can also be calculated.

Proof of Concept

Context: BasicOrderFulfiller.sol#L118-L128

The function `_validateAndFulfillBasicOrder()` has a hardcoded value of `FiveWords` to calculate the distance between the location of `considerationToken` and the `offerToken` in the `struct BasicOrderParameters`.

```
    function _validateAndFulfillBasicOrder(
        ...
        assembly {
                // Determine if offered item type == additional reci
                let offerTypeIsAdditionalRecipientsType := gt(route,

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

    struct BasicOrderParameters {
        address considerationToken;
        uint256 considerationIdentifier;
        uint256 considerationAmount;
        address payable offerer;
        address zone;
        address offerToken;
      ...
    }
```

Recommended Mitigation Steps

Calculate the distance between `considerationToken` and the `offerToken` and use that instead of `FiveWords`:

```
uint256 constant BasicOrder_considerationToken_cdPtr = 0x24;
uint256 constant BasicOrder_offerToken_cdPtr = 0xc4;

+uint256 constant BasicOrder_Distance_oc_cdPtr = BasicOrder_offe

- mul(offerTypeIsAdditionalRecipientsType, FiveWords)
+ mul(offerTypeIsAdditionalRecipientsType, BasicOrder_Distance_c
```

Note: this suggestion could also be used for the assignment of `conduitKey`. See BasicOrderFulfiller.sol#L165-L170 and BasicOrderFulfiller.sol#L1026-L1033

[05] Helpers should have their expectations explained in NatSpec/comments

Certain functions with `unchecked` blocks or other "exotic logic" have assumptions for certain input values. These assumptions are not documented well. It makes it error prone to validate against possible inputs and/or expected behaviour.

Examples:

1. is `_locateCurrentAmount` / `_applyFraction` which has a strong dependency on `duration != 0` and `startTime < endTime` (and `!=`). There are other cases too.
2. In AmountDeriver.sol#L13. The correct word is "interpolation", not "extrapolation". Similarly at other places where the word is used.
3. In AmountDeriver.sol#L114, there are assumptions about the range of amounts supported in the protocol. This need to be documented well. Protocols such as Uniswap makes such assumptions explicit. More specifically, there are overflow issues when `value >= type(uint).max / type(uint120).max` Around `2**136`.

Proof of Concept

Context: lib/AmountDeriver.sol#L33, lib/AmountDeriver.sol#L138

Recommended Mitigation Steps

Document these assumptions.

[06] `ConduitTransfer` identifier field can be dirty and unused

There are two external entry points `execute(ConduitBatch1155Transfer[] calldata batchTransfers)` and `executeWithBatch1155(ConduitTransfer[] calldata standardTransfers, ConduitBatch1155Transfer[] calldata batchTransfers)` which trigger the internal `_transfer(...)` function. It works off this data structure:

```
    struct ConduitTransfer {
        ConduitItemType itemType;
        address token;
        address from;
        address to;
        uint256 identifier;
        uint256 amount;
    }
```

The `_transfer` function only supports the ERC-20/ERC-721/ERC-1155 item types, where it ensures that `amount == 1` for ERC-721, but allows `identifier` to be anything for ERC-20 transfers.

It could be:

1. used to create multiple transactions which have identical outcomes (since the field is ignored)
2. misused to masquerade an ERC-20 transfer to look more similar to an ERC-721 transfer.

This problem is similar to the `SpentItem/ReceivedItem fields can be dirty and unused` issue and can be triggered through that transaction, but since the `Conduit` is a general purpose feature and could be used and triggered by other contracts too.

We believe that can have severe risks in the future. But it is hard to argue about the severity of this currently due to lack of clarity on what kinds of applications would be built on top of conduits.

Proof of Concept

Context: Conduit.sol#L174, Conduit.sol#L52, Conduit.sol#L117

Recommended Mitigation Steps

Insert check that `item.identifier == 0` for `ConduitItemType.ERC20`.

[07] Inconsistent way to return values in `_verifyTime()` and `_verifyOrderStatus`

The functions `_verifyTime()` and `_verifyOrderStatus` use two different ways to return a value. For consistency its better to use the same way every time.

Proof of Concept

Context: Verifiers.sol#L37-L55, Verifiers.sol#L102-L139

```
function _verifyTime( ... ) internal view returns (bool valid) {
    ...
    if ( ... ) {
        ...
        // Return false as the order is invalid.
        return false; // method 1
    }
    // Return true as the order time is valid.
    valid = true; // method 2
}
```

```
function _verifyOrderStatus( ... ) internal pure returns (bool v
    ...
    if ( ... ) {
        // Return false as the order status is invalid.
        return false; // method 1
    }
    // Return true as the order status is valid.
```

```
        valid = true; // method 1
    }
```

Recommended Mitigation Steps

Consider changing the code in the functions `_verifyTime()` and `_verifyOrderStatus` in the following way:

```
    - valid = true;
    + return true;
```

[08] `_callConduitUsingOffsets` depends on compiler behaviour for inter-assembly-block cleanup

The function `_callConduitUsingOffsets` depends on the compiler not to use the scratch space between assembly blocks. it also performs some Solidity function calls between the two blocks.

A compiler behaviour change would render calling conduits always failing.

Furthermore this function depends on `_revertWithReasonIfOneIsReturned` not disturbing anything (see another relevant issue by us).

Proof of Concept

Context: [Executor.sol#L498]

```
            bool success;

            // call the conduit.
            assembly {
                // Ensure first word of scratch space is empty.
                mstore(0, 0)

                // Perform call, placing first word of return data i
                success := call(
                    gas(),
                    conduit,
```

```
                0,
                callDataOffset,
                callDataSize,
                0,
                OneWord
            )
        }

        // <--- If the compiler changes scratch space after this

        // If the call failed...
        if (!success) {
            // Pass along whatever revert reason was given by th
            _revertWithReasonIfOneIsReturned();

            // Otherwise, revert with a generic error.
            revert InvalidCallToConduit(conduit);
        }

        // Ensure that the conduit returned the correct magic va
        bytes4 result;
        assembly {
            // Take value from scratch space and place it on the
            result := mload(0)
        }

        // Ensure result was extracted and matches EIP-1271 magi
        if (result != ConduitInterface.execute.selector) {
            revert InvalidConduit(conduitKey, conduit);
        }
```

Recommended Mitigation Steps

1. Load the scratch space in the first assembly block
2. Since this function seems to perform “return-magic-detection”, one could consider replacing most of this with `_doesNotMatchMagic`

[09] `Assertions.sol` is missing an assertion about bounds for array length

Context: [Assertions.sol#L129-L144]

So far, the impact seems low and we are not able to craft an exploit.

```
    validOffsets := and(
        validOffsets,
        eq(
            // Load signature offset from calldata 0x244.
            calldataload(BasicOrder_signature_cdPtr),
            // Derive expected offset as start of recipients + len *
            add(
                BasicOrder_signature_ptr,
                mul(
                    // Additional recipients length at calldata 0x2(
                    calldataload(
                        BasicOrder_additionalRecipients_length_cdPtr
                    ),
                    // Each additional recipient has a length of 0x4
                    AdditionalRecipients_size
                )
            )
        )
```

The above code checks for the following, let `len` represent the length of the `additionalRecipients` array, then it checks that `len * 64 + 0x260 == calldataload(0x244)`. Where all the arithmetic is in EVM. This however does not check for overflow of `len * 64`. It's possible to craft malicious calldata that would satisfy this condition by overflowing on the multiplication. We need to find `x` such that `mul(x, 64) = mul(len, 64)` in EVM arithmetic. The values of `x` can be `len + y` where `y` is in the set `{2**250, 2**251, ..., 2**255}`. These are also the only such values.

This would create problems whenever the value `BasicOrder_additionalRecipients_length_cdPtr` is used to do read from calldata.

Note: the solidity compiler would add the check `len < 2**64` for high level data.
Note: the issues is similar to some known bugs in past versions of solc. Look for bug descriptions with "overflow" in bugs.json.

Proof of Concept

The following addition to the test would revert, but the reverts happen later in the codebase, likely due to OOG as the value gets used in a for-loop in BasicOrderFulfiller.sol#L611-L13.

The test changes the length of an empty array to `2**255`, everything else in the `calldata` remaining the same. The invariant `0 * 64 + 0x260 == 0x260 == 2**255 * 64 + 0x60` is true in EVM arithmetic.

```
modified   test/index.js
@@ -15945,6 +15945,25 @@ describe(`Consideration (version: ${VER
         ).to.be.revertedWith("InvalidBasicOrderParameterEncodir
       });

+      it("Malicious calldata", async () => {
+        console.log(`Good data: ${calldata}`)
+        console.log(`calldata[0x284:0x2a4]: ${calldata.slice(2
+        const badData = [calldata.slice(0, 2 * 0x284 + 2), "f0(
+          ""
+        ];
+        console.log(`Bad data: ${badData}`)
+        expect(badData.length).to.eq(calldata.length);
+
+        await expect(
+          buyer.sendTransaction({
+            to: marketplaceContract.address,
+            data: badData,
+            value,
+          })
+        ).to.be.revertedWith("InvalidBasicOrderParameterEncodir
+      });
+
       it("Reverts if additionalRecipients has non-default offse
         const badData = [
```

Recommended Mitigation Steps

Consider adding a check for `calldataload(BasicOrder_additionalRecipients_length_cdPtr) < 2**64`, similar to what `solc` generates. Alternatively, the number `2**64` can be decreased further, depending on a realistic upper bound for `additionalRecipients`.

[10] LowLevelHelpers enforce a stricter ABI standard for `returndatasize`

Context: [LowLevelHelpers.sol#L110]

```
            assembly {
                // Only put result on stack if return data is exactl
                if eq(returndatasize(), OneWord) {
```

This is not compliant with the ABI standard, as the ABI standard allows for returning more data than needed. The proper one should `if iszero(lt(returndatasize(), OneWord))`.

This is currently used in

1. `_assertValidEIP1271Signature` : to check the returnvalue `bytes4 magicValue` of `isValidSignature`.
2. `_assertIsValidOrderStaticcallSuccess` for checking `ZoneInterface` returnvalue.

The ABI standard allows for extra data everywhere. That is, given a correct 32-byte value, having more data at the end is valid. Therefore, this function returns `false` for complaint contracts.

However, most contracts wouldn't return more than 32-bytes when only 32-bytes are needed. A notable exception is some versions of proxy contracts in Vyper. This proxy always returned 128 bytes, with any extra data padded with zeros (source). This has caused issues with SolMate's `SafeTransferLib`, leading to DOS issues on chain (source).

This is at least a low severity issue, and may even be considered Medium if there are known EIP1271 wallets in existence with the aforementioned issue. Note: for ERC20 tokens, there are known tokens with the problem (certain Curve tokens). Considering that this is a trivial fix, it's better to be on the side of caution and make the above change.

Proof of Concept

For a simple proof of concept, consider a minimal proxy contract that does `return(0, 0x1000)` (instead of the usual `return(0, returndatasize())`) where the 'implementation' is a EIP1271 wallet contract. The low level helper would revert when verifying the `_assertValidEIP1271Signature`, whereas a high level solidity implementation would succeed.

Recommended Mitigation Steps

Change the strict equality check to a `>=` check.

[11] `_assertValidSignature` allows malleable `secp256k1` signatures

The `ecrecover()` call here does not verify `0 < s < secp256k1.n ÷ 2 + 1`.

This restriction was introduced Homestead (EIP-2), but left the precompile unchanged. Most libraries, such as OpenZeppelin, perform this check.

There does not seem to be an immediate risk in the current use cases, because an order can be verified only once and its hash is stored. These places are `_validateBasicOrderAndUpdateStatus`, `_validateOrderAndUpdateStatus` and `validate`.

Proof of Concept

Context: SignatureVerification.sol#L91

Recommended Mitigation Steps

Perform the check.

[12] The memory cost calculation logic `_revertWithReasonIfOneIsReturned` is duplicated in multiple places

`_revertWithReasonIfOneIsReturned` implements a memory cost calculation logic. This is duplicated in four places: `_performERC20Transfer`,

`_performERC721Transfer`, `_performERC1155Transfer`, `_performERC1155BatchTransfer`.

Several slight issues were identified with this function and those may or may not be present in five places in total.

Proof of Concept

Context: LowLevelHelper.sol#L46, TokenTransferrer.sol#L79, TokenTransferrer.sol#L259, TokenTransferrer.sol#L393, TokenTransferrer.sol#L639

Recommended Mitigation Steps

Reduce code duplication and risk of differences between them.

[13] Accumulator code depends on memory usage

The accumulator is used in `BasicOrderFulfiller`, `OrderFulfiller` and `OrderCombiner`. It is a variable length array, but does not have a properly allocated memory space. It can be grown using `_insert` and `_trigger` will "flush" it.

The problem is it is placed at the free memory pointer, but the pointer is not increment if it would grow beyond its initial capacity. The initial capacity is `AccumulatorDisarmed` aka 32. In the use cases it seems at most 2 entries are inserted.

Instead of relying that this memory space is never overwritten, it may make sense pre-allocating a fixed structure. The current implementation can break on compiler upgrades or changes in the function layouts.

Proof of Concept

Context: `BasicOrderFulfiller`, `OrderFulfiller`, `OrderCombiner`, `_insert`, `_trigger`

Recommended Mitigation Steps

Avoid using the non-memory allocating accumulator design.

[14] Doing a basic order twice gives a misleading error message

When doing a basic order twice a misleading error message is given: `OrderPartiallyFilled()`.

Proof of Concept

Context: OrderValidator.sol#L48-L74, Verifiers.sol#L102-L139

When doing a basic order, the function `_validateBasicOrderAndUpdateStatus()` is called, which uses `_verifyOrderStatus()` to verify the order hasn't been used before. After this function it sets `_orderStatus[orderHash].numerator = 1;` to indicate the `orderHash` has been used. When trying to reuse the same order, `_verifyOrderStatus()` correctly `revert`s. However it uses the `revert` message `OrderPartiallyFilled()` which is not correct because the order has been fully filled before.

```
contract OrderValidator is Executor, ZoneInteraction {
    function _validateBasicOrderAndUpdateStatus( ... ) ... {
        ...
        _verifyOrderStatus(
            orderHash,
            orderStatus,
            true, // Only allow unused orders when fulfilling ba
            true // Signifies to revert if the order is invalid.
        );
     ...
    _orderStatus[orderHash].numerator = 1;
    ...
    }
}
```

```
contract Verifiers is Assertions, SignatureVerification {
    function _verifyOrderStatus(..., bool onlyAllowUnused, ... )
        ...
        if (orderStatus.numerator != 0) {  // the second time nu
            if (onlyAllowUnused) {   // true
                revert OrderPartiallyFilled(orderHash);  // misl
                ...
```

```
                }
            }
        }
    }
     revert OrderPartiallyFilled(orderHash);
```

Recommended Mitigation Steps

Consider giving a different error message in `_verifyOrderStatus()` when a basic order is executed twice.

[15] Orders with moving price and `endTime = type(uint).max` can never be fulfilled

It is quite realistic that many users will use `type(uint).max` as `infinity` when setting the `endTime` of their orders so that it stays open indefinitely. However the math `(startAmount * remaining) + (endAmount * elapsed) + extraCeiling` in https://github.com/ProjectOpenSea/seaport/blob/49799ce156d979132c9924a739ae45a38b39ecdd/contracts/lib/AmountDeriver.sol#L57 will almost always revert in that case since `remaining` will be very large and this block of code is checked Solidity.

Proof of Concept

Context: Seaport.sol

Use the new test below and run `forge test -m testAdvancedPartialMaxEndTime -vvvv` :

```
    function testAdvancedPartialMaxEndTime() public {
            FuzzInputs memory inputs = FuzzInputs(
                    0,
                    address(0),
                    bytes32(0),
                    0,
                    0,
                    [uint120(10), 20, 30],
                    false,
```

```
                1,
                1
        );

        _testAdvancedPartialMaxEndTime(
                Context(consideration, inputs, 10, 10)
        );
}

function _testAdvancedPartialMaxEndTime(
        Context memory context
) internal resetTokenBalancesBetweenRuns {
        bytes32 conduitKey = context.args.useConduit
                ? conduitKeyOne
                : bytes32(0);

        uint startAmount = 10;
        uint endAmount = 20;

        test1155_1.mint(
                alice,
                context.args.tokenId,
                endAmount
        );

        _configureOfferItem(
                ItemType.ERC1155,
                context.args.tokenId,
                startAmount,
                endAmount
        );
        _configureEthConsiderationItem(
                alice,
                10
        );

        OrderParameters memory orderParameters = OrderParameters
                address(alice),
                context.args.zone,
                offerItems,
                considerationItems,
                OrderType.PARTIAL_OPEN,
                block.timestamp, // startTime
                type(uint).max, // endTime
                context.args.zoneHash,
                context.args.salt,
```

```
                conduitKey,
                considerationItems.length
        );

        OrderComponents memory orderComponents = getOrderCompone
                orderParameters,
                context.consideration.getNonce(alice)
        );

        bytes32 orderHash = context.consideration.getOrderHash(c

        bytes memory signature = signOrder(
                context.consideration,
                alicePk,
                orderHash
        );

        delete offerItems;
        delete considerationItems;

        AdvancedOrder memory advancedOrder = AdvancedOrder(
                orderParameters,
                1,
                1,
                signature,
                ""
        );

        context.consideration.fulfillAdvancedOrder{
                value: 10
        }(advancedOrder, new CriteriaResolver[](0), bytes32(0));
        (, , uint256 totalFilled, uint256 totalSize) = context
                .consideration
                .getOrderStatus(orderHash);
    }
```

Recommended Mitigation Steps

Need to find a way to compute the linear interpolation without overflows. For `(a + b) / 2`, the trick is `a / 2 + b / 2 + (a & b & 1)`: https://devblogs.microsoft.com/oldnewthing/20220207-00/?p=106223

Need to find a similar trick for `(x * a + y * b) / (x + y)`.

Worst case it can also be a front end fix.

[16] `_prepareBasicFulfillmentFromCalldata` overwrites memory extensively

The `_prepareBasicFulfillmentFromCalldata` function prepares ABI encoded data at fixed location in memory (starting at `0x80` and writing as high as `0x1e0+0x20`).

The overwritten memory does not seem to be saved, and this function has multiple assembly blocks (probably for avoiding `stack too deep errors` ?). The only field restored is the zero slot.

The only use in Seaport is the `Consideration.fulfillBasicOrder` → `_validateAndFulfillBasicOrder` → `_prepareBasicFulfillmentFromCalldata` call chain, where this does not seem to be a problem, because they are not allocating memory prior to entering this function (according to displaying the free memory pointer in the test suite).

This would become a big issue if some functions would allocate memory. It is also very dependent on the compiler version.

Proof of Concept

Context: Consideration.sol#L76, BasicOrderFulfiller.sol#L70, BasicOrderFulfiller.sol#L325

Recommended Mitigation Steps

Consider restoring the dirty memory.

[17] Parameters passed to `fulfillBasicOrder()` insufficiently checked

The parameters that are passed to function `fulfillBasicOrder()` aren't sufficiently checked. Due to some lucky circumstances this doesn't lead to issues. However it is safer to do additional checking.

Proof of Concept

Context:
[BasicOrderFulfiller.sol#L70-L295](), [Consideration.sol#L76-L84]()

The parameters are passed from `fulfillBasicOrder()` to `_validateAndFulfillBasicOrder()`. As these `parameters` are `calldata`, no bounds checking is done.

Assume the field for `BasicOrder_basicOrderType` contains the value of 6 * 4. Then the `route` will be 6, which is an out of bounds value. However while in assembly this is not detected. When calculating `receivedItemType`, it will get a value of 4, which is a valid value (`ERC721_WITH_CRITERIA`). However if this would be used, the rest of the logic isn't prepared for this value.

Luckily after the `assembly` code, `route` is evaluated in Solidity. At that moment Solidity catches the out of bound value and reverts. However if the code would be optimized with more assembly then this would not have been caught.

```
contract Consideration is ConsiderationInterface, OrderCombiner
    function fulfillBasicOrder(BasicOrderParameters calldata par
        ...
        fulfilled = _validateAndFulfillBasicOrder(parameters);
    }
}
contract BasicOrderFulfiller is OrderValidator {
    function _validateAndFulfillBasicOrder(  BasicOrderParameter
        ...   // as the parameters are calldata, no checks on ou
        BasicOrderRouteType route;
        ItemType receivedItemType;
        ...
        assembly {
            ...
            route := div(calldataload(BasicOrder_basicOrderType_
            ...
            receivedItemType := add( // if route == 6, then rece
                mul(sub(route, 2), gt(route, 2)), // (6-2) * 1 =
                eq(route, 2)                      // + 0 == 4   /
        )
        ...
        if (additionalRecipientsItemType == ItemType.NATIVE) {
```

```
            ...
            } else {
                ...
                if (route == BasicOrderRouteType.ERC20_TO_ERC721) {
                    ...
                }
            }
        }
    }
```

Recommended Mitigation Steps

In the `assembly` code, check that `route` is within range.

[18] `startAmount` and `endAmount` being different would severely limit partial orders

Let n and d represent `numerator` and `denominator` respectively, where $n \le d$. Similarly, s and e represent `startAmount` and `endAmount`. Assume that $s \neq e$ and for the sake of simplicity let's assume that n and d are in reduced form (coprime, i.e., $\operatorname{gcd}(n, d) = 1$).

Then, the following conditions have to be true for an order (here $a \mid b$ means a divides b):

- $d \mid s$: AmountDeriver.sol#L155.
- $d \mid e$: AmountDeriver.sol#L156.

This severely limits the possibilities of an order. For example, if $\operatorname{gcd}(s, e) = 1$, then a strict partial order is impossible—only a full fill (`1 / 1`) would ever get past such checks.

Proof of Concept

Context: Seaport.sol

Alice places a partial order with `startAmount = 1000` and `endAmount = 2001`. Because `1000` and `2001` are coprime, such an order can only be fully filled.

Recommended Mitigation Steps

Partially fillable orders with `gcd(startAmount, endAmount) == 1` should ideally be disallowed. This may be done at the frontend to simplify the code.

[19] SafeTransferFrom: `transferFrom` to precompiles may succeed, a deviation from OZ's implementation

Context: [TokenTransferrer.sol#L72]

```
// If the token has no code or the transfer failed:
// Equivalent to `or(iszero(success), iszero(extcodesize(token))
// but after it's inverted for JUMPI this expression is cheaper.
if iszero(and(iszero(iszero(extcodesize(token))), success)) {
```

The `extcodesize` check is only done here. In contrast, the Openzeppelin implementation would do the `extcodesize` check before calling the function. Here's where this function and OZ's safetransfer would deviate: the `token` address is a precompile (so `extcodesize() == 0`), but it can still return data. For the same calldata `abi.encodeWithSelector(ERC20.transferFrom.selector, from, to, amount)`:

1. it should return at least 32 bytes, (this is easy, for example the identity precompile at address `4`).
2. the first 32 bytes of the `returndata` is 1. This looks a bit tricky to achieve, but perhaps with the right parameters, the `modexp` precompile should do the trick?

There were some concerns of being able to spoof this function source.

It's also arguable that this is really a problem. Since `0.8.10`, the high level call `ERC20.transferFrom(...)` would skip the `extcodesize` check because it's always followed up by a `abi.decode(...)`. But `transferFrom` is a bit of a grey area because it needs compatibility with non-compliant ERC20 tokens like USDT.

Proof of Concept

It is hard to make a proof of concept for this. In the future, a new precompile may change this.

Recommended Mitigation Steps

Consider adding an `extcodesize()` check before `call`. However, this adds an extra `100` gas and we can understand if the protocol decides to not implement this. :)

[20] The gas computation for memory expansion rounds down instead of rounding up

Context: [LowLevelHelpers.sol#L54]

```
if returndatasize() {
    // Ensure that sufficient gas is available to cc
    // while expanding memory where necessary. Start
    // the word size of returndata and allocated mem
    let returnDataWords := div(returndatasize(), One
```

This rounds down the number of words (for example, if `returndatasize() = 31` the correct number of words is 1). The number of words is defined rounded up in EVM.

For a precise calculation, it should be `div(add(returndatasize(), 31), 32)`. A typical routine in internal compiler code. Here is how the Solidity compiler does it.

Proof of Concept

The above issue can affect the gas computation for `returndata`, although unlikely to the point that it is severe as the computation is still making some assumptions about `msize`.

Recommended Mitigation Steps

Replace the rounding from down to up.

[21] `TokenTransferrer._performERC1155BatchTransfers` leaves corrupted memory

The function will overwrite memory starting from the `0x20` offset at least `0x104` bytes (`BatchTransfer1155Params_data_length_basePtr * idsLength`).

In case of successful completion, it will only "restore" the free memory pointer to the starting value (`mstore(FreeMemoryPointerSlot, DefaultFreeMemoryPointer)`), which is likely invalid, and will leave the zero slot and any potential user memory area dirty.

Proof of Concept

This function is only used in the `Conduit` in two places, `executeBatch1155` and `executeWithBatch1155` , e.g.:

```
function executeBatch1155(
    ConduitBatch1155Transfer[] calldata batchTransfers
) external override returns (bytes4 magicValue) {
    // Ensure that the caller has an open channel.
    if (!_channels[msg.sender]) {
        revert ChannelClosed();
    }

    // Perform 1155 batch transfers.
    _performERC1155BatchTransfers(batchTransfers);

    // Return a magic value indicating that the transfers we
    magicValue = this.executeBatch1155.selector;
}
```

The only statement after it is returning a value type, and both of these functions are marked external, so likely this is not causing any problems the way it is used currently, but the library function is unsafe in itself.

Recommended Mitigation Steps

Properly restore the corrupted memory area, similar to what `_performERC1155Transfer` is doing.

[22] `_triggerIfArmed()` done outside of reentrancy guards

The function `fulfillBasicOrder()` of the reference implementation is entirely protected by the reentrancy guard. However in the production implementation, the call to `_triggerIfArmed()` isn't protected by the reentrancy guard. As `_triggerIfArmed()` does external calls this doesn't seem logical, however the risk seems low.

For comparison: the comparable function `_validateAndFulfillAdvancedOrder()` is protected end to end by a reentrancy guard, which protects the call to `_applyFractionsAndTransferEach()` and thus the call to `_triggerIfArmed(accumulator);`

Proof of Concept

Context: ReferenceConsideration.sol#L83-L93, Consideration.sol#L76-L84, BasicOrderFulfiller.sol

Reference code:

```
contract ReferenceConsideration is ConsiderationInterface, Refer
    ...
    function fulfillBasicOrder(BasicOrderParameters calldata par
        ...
    }
    ...
}
```

Production code:

```
contract Consideration is ConsiderationInterface, OrderCombiner
    ...
    function fulfillBasicOrder(BasicOrderParameters calldata par
        fulfilled = _validateAndFulfillBasicOrder(parameters);
    }

    function _validateAndFulfillBasicOrder( ... ) ... {
        ...
        _prepareBasicFulfillmentFromCalldata( ... ); // sets ree
        ...
```

```
            _transferEthAndFinalize(... ) --or-- _transferERC20AndFi
            ...
            _triggerIfArmed(accumulator);  // not protected by reent
            ...
        }

        function _prepareBasicFulfillmentFromCalldata(...) ... {
            // Ensure this function cannot be triggered during a ree
            _setReentrancyGuard();
            ...
        }
         function _transferEthAndFinalize(...) ... {
            ...
            // Clear the reentrancy guard.
            _clearReentrancyGuard();
        }
      function _transferERC20AndFinalize(...) ... {
            ...
            // Clear the reentrancy guard.
            _clearReentrancyGuard();
       }
    }
```

Recommended Mitigation Steps

Do the `_clearReentrancyGuard();` after the call to `_triggerIfArmed(accumulator);`

[23] Deviations between Solidity compiler's checks and seaport's checks in `validateOrderParameters`

Context: [Assertions.sol#L105]

Some comments on comparison between code produced by solidity and this:

1. If there is a parameter `BasicOrderParameters calldata`, the compiler generates the following checks:
    1. `calldatasize() < 2**64`.
    2. `calldataload(4) < 2**64`. (Check if the initial offset is too big)

    3. `calldatasize() - offset >= 0x244`.

2. The ABI encoder V2 has additional checks on whether `calldata` is properly clean. The compiler only does this checks when a value is read (a high level read; assembly doesn't count). If you want to be complaint, then the values will need to be checked for sanity. For example, an `address` type should not have dirty higher order bits. For example, for `considerationToken`.
3. This does not check for upper bounds of length of the array `additionalRecipients`. The compiler typically checks if length is `< 2**64`. Similarly, for `bytes signature`. The length checks are surprisingly needed in general, otherwise some offset calculations can overflow and read values that it is not supposed to read. This can be used to fool some checks. Mentioned below.
4. Both `additionalRecipents` and `signature` are responsible for at least 1 word each in `calldata` (at least length should be present). The compiler checks this. But is likely missing here.

    1. `calldataEncodedTailSize`
    2. the check for tail size

5. The compiler checks that the length of the two dynamic arrays (appropriately scaled) + offsets wouldn't be past `calldatasize()`. (Note: reading past `calldatasize()` would return 0).

Recommended Mitigation Steps

Document the differences. Consider adding additional checks, if the differences need to be accounted. See a related issue regarding overflowing length, which ideally needs to be fixed.

0xleastwood (judge) commented:

> This report and its merged issues* highlight several limitations which are informative to the Opensea team. This report is of high quality and is deserving of the best score. I consider all issues raised to be valid.

> *Merged issues: #108, 156, 176, 195, and 205.

Gas Optimizations

For this contest, 45 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: shung, OriDabush, IIIIIII, Spearbit, cmichel, 0x1f8b, NoamYakov, djxploit, 0xalpharush, Chom, Czar102, hickuphh3, 0x29A, sirhashalot, csanuragjain, ming, gzeon, MaratCerby, zkhorse, zer0dot, defsec, Hawkeye, ignacio, joestakey, MiloTruck, rfa, oyc_109, cccz, sashik_eth, ilan, kaden, sach1r0, TomJ, twojoy, ellahi, TerrierLover, asutorufos, delfin454000, hake, mayo, peritoflores, RoiEvenHaim, Tadashi, *and* foobar.

Table of Contents

- [G-11] Bytes constants are more efficient than string constants
- [G-12] An array's length should be cached to save gas in for-loops
- [G-13] Increments can be unchecked
- [G-14] No need to explicitly initialize variables with default values
- [G-15] `abi.encode()` is less efficient than `abi.encodePacked()`

[G-01] Cheap Contract Deployment Through Clones

See `@audit` tag:

```
contracts/conduit/ConduitController.sol:
   91:         new Conduit{ salt: conduitKey }(); //@audit gas:
```

There's a way to save a significant amount of gas on deployment using Clones: https://www.youtube.com/watch?v=3Mw-pMmJ7TA .

This is a solution that was adopted, as an example, by Porter Finance. They realized that deploying using clones was 10x cheaper:

- https://github.com/porter-finance/v1-core/issues/15#issuecomment-1035639516
- https://github.com/porter-finance/v1-core/pull/34

I suggest applying a similar pattern, here with a `cloneDeterministic` method to mimic the current `create2`

[G-02] `ConduitController.sol#createConduit()` : Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it

To help the optimizer, declare a `storage` type variable and use it instead of repeatedly fetching the reference in a map or an array.

The effect can be quite significant.

As an example, instead of repeatedly calling `someMap[someIndex]`, save its reference like this: `SomeStruct storage someStruct = someMap[someIndex]` and use it.

Affected code (check the `@audit` tags):

```
File: ConduitController.sol
- 94:          _conduits[conduit].owner = initialOwner;  //@audit
+ 93:          ConduitProperties storage _conduitProperties = _co
+ 94:          _conduitProperties.owner = initialOwner;
95:
96:         // Set conduit key used to deploy the conduit to ena
- 97:          _conduits[conduit].key = conduitKey;//@audit gas:
+ 97:         _conduitProperties.key = conduitKey;
```

Notice that this optimization already exists in the solution:

```
File: ConduitController.sol
129:          // Retrieve storage region where channels for the c
130:          ConduitProperties storage conduitProperties = _cond
```

[G-03] `ConduitController.sol#acceptOwnership()` : Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it

This optimization is similar to the one explained above in G-02.

Instead of repeatedly fetching the storage region, consider declaring and using a storage variable here (see `@audit` tags):

```
File: ConduitController.sol
232:     function acceptOwnership(address conduit) external over
...
- 237:          if (msg.sender != _conduits[conduit].potentialOwn
+ 236:          ConduitProperties storage _conduitProperties = _c
+ 237:          if (msg.sender != _conduitProperties.potentialOwn
...
```

```
- 246:          delete _conduits[conduit].potentialOwner;//@audit
+ 246:          delete _conduitProperties.potentialOwner;
...
249:          emit OwnershipTransferred(
250:              conduit,
- 251:              _conduits[conduit].owner, //@audit gas: shoul
+ 251:              _conduitProperties.owner,
252:              msg.sender
253:          );
...
- 256:          _conduits[conduit].owner = msg.sender; //@audit g
+ 256:          _conduitProperties.owner = msg.sender;
```

[G-04] `OrderValidator.sol#_validateBasicOrderAndUpdateStatus()` : Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it

This optimization is similar to the one explained here.

Instead of repeatedly fetching the storage region, consider declaring and using a storage variable here (see `@audit` tags):

```
File: OrderValidator.sol
48:     function _validateBasicOrderAndUpdateStatus(
...
- 70:          _orderStatus[orderHash].isValidated = true; //@auc
+ 69:          OrderStatus storage _orderStatusStorage = _orderSt
+ 70:          _orderStatusStorage.isValidated = true;
...
- 72:          _orderStatus[orderHash].numerator = 1;//@audit gas
- 73:          _orderStatus[orderHash].denominator = 1;//@audit g
+ 72:          _orderStatusStorage.numerator = 1;
+ 73:          _orderStatusStorage.denominator = 1;
74:     }
```

[G-05] `OrderValidator.sol#_validateBasicOrderAndUpdateStatu`

`s()` : avoid an unnecessary SSTORE by not writing a default value

The following line is not needed, as it's writing to storage a default value:

```
File: OrderValidator.sol
71:         _orderStatus[orderHash].isCancelled = false;//@audit
```

Consider removing this line completely.

[G-06] `OrderCombiner.sol` : `++totalFilteredExecutions` costs less gas compared to `totalFilteredExecutions += 1`

For a `uint256 i` variable, the following is true with the Optimizer enabled at 10k:

- `i += 1` is the most expensive form
- `i++` costs 6 gas less than `i += 1`
- `++i` costs 5 gas less than `i++` (11 gas less than `i += 1`)

Consider replacing `totalFilteredExecutions += 1` with `++totalFilteredExecutions` here:

```
lib/OrderCombiner.sol:490:                    totalFilteredExecu
lib/OrderCombiner.sol:515:                    totalFilteredExecu
lib/OrderCombiner.sol:768:                    totalFilteredExecu
```

[G-07] `OrderCombiner.sol` : `--maximumFulfilled` costs less gas compared to `maximumFulfilled--`

For a `uint256 i` variable, the following is true with the Optimizer enabled at 10k:

- `i -= 1` is the most expensive form
- `i--` costs 11 gas less than `i -= 1`
- `--i` costs 5 gas less than `i--` (16 gas less than `i -= 1`)

Consider replacing `maximumFulfilled--` with `--maximumFulfilled` here:

```
lib/OrderCombiner.sol:229:                maximumFulfilled--;
```

[G-08] `FulfillmentApplier.sol#_applyFulfillment()` : Unchecking arithmetics operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an `unchecked` block:
https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

I suggest wrapping with an `unchecked` block here (see `@audit` tags for more details):

```
File: FulfillmentApplier.sol
090:         if (considerationItem.amount > execution.item.am
...
097:             advancedOrders[targetComponent.orderIndex]
098:                 .parameters
099:                 .consideration[targetComponent.itemIndex
100:                 .startAmount = considerationItem.amount
...
104:         } else {
...
109:             advancedOrders[targetComponent.orderIndex]
110:                 .parameters
111:                 .offer[targetComponent.itemIndex]
112:                 .startAmount = execution.item.amount - c
113:         }
```

[G-09] `/reference` : Unchecking arithmetics operations that can't underflow/overflow

This is similar to the optimization above, except that here, contracts under `/reference` are just readable versions of the real contracts to be deployed.

The following lines should be `unchecked`, please check that this is the case in their corresponding assembly code:

- reference/lib/ReferenceBasicOrderFulfiller.sol:

```
      842              if (additionalRecipientAmount > etherRema
      843                  revert InsufficientEtherSupplied();
      844              }
      ...
 843  853:             etherRemaining -= additionalRecipientAmou
```

```
      865          if (etherRemaining > amount) {
      866              // Transfer remaining Ether to the caller
 865  867:             _transferEth(payable(msg.sender), etherRe
 866  868          }
```

- reference/lib/ReferenceFulfillmentApplier.sol:

```
       99          // If total consideration amount exceeds the
      100          if (considerationItem.amount > execution.item
      ...
      107              ordersToExecute[targetComponent.orderInde
      108                  .receivedItems[targetComponent.itemIn
 100  109:                .amount = considerationItem.amount -
 101  ...
 102  113          } else {
 103  ...
 104  118              ordersToExecute[targetComponent.orderInde
 105  119                  .spentItems[targetComponent.itemIndex
 100  120:                .amount = execution.item.amount - con
 101  121          }
```

```
      189          // If no available order was located...
      190          if (nextComponentIndex == 0) {
      191              // Return with an empty execution element
```

```
192                 // prettier-ignore
193                 return Execution(
194                     ReceivedItem(
195                         ItemType.NATIVE,
196                         address(0),
197                         0,
198                         0,
199                         payable(address(0))
200                     ),
201                     address(0),
202                     bytes32(0)
203                 );
204             }
205
206             // If the fulfillment components are offer co
207             if (side == Side.OFFER) {
208                 // Return execution for aggregated items
209                 // prettier-ignore
210                 return _aggregateValidFulfillmentOfferIte
211                     ordersToExecute,
212                     fulfillmentComponents,
190 213:                nextComponentIndex - 1
191 214                 );
192 215             } else {
193 216                 // Otherwise, fulfillment components are
194 217                 // components. Return execution for aggre
195 218                 // the fulfiller.
196 219                 // prettier-ignore
197 220                 return _aggregateConsiderationItems(
198 221                     ordersToExecute,
199 222                     fulfillmentComponents,
190 223:                nextComponentIndex - 1,
191 224                     fulfillerConduitKey
192 225                 );
193 226             }
```

- reference/lib/ReferenceOrderCombiner.sol:

```
629                     if (item.amount > etherRemaining) {
630                         revert InsufficientEtherSupplied(
631                     }
632
633                     // Reduce ether remaining by amount.
```

```
629    634:                 etherRemaining -= item.amount;
```

- reference/lib/ReferenceOrderFulfiller.sol:

```
       220                   if (amount > etherRemaining) {
       221                       revert InsufficientEtherSuppl
       222                   }
       223                   // Reduce ether remaining by amou
220    224:                   etherRemaining -= amount;
```

```
       273                   if (amount > etherRemaining) {
       274                       revert InsufficientEtherSuppl
       275                   }
       276                   // Reduce ether remaining by amou
273    277:                   etherRemaining -= amount;
```

- reference/lib/ReferenceOrderValidator.sol:

```
       220               if (filledNumerator + numerator > denomin
       221                   // Reduce current numerator so it + s
220    222:                   numerator = denominator - filledNumer
221    223               }
```

[G-10] `OR` conditions cost less than their equivalent `AND` conditions ("NOT(something is false)" costs less than "everything is true")

Remember that the equivalent of `(a && b)` is `!(!a || !b)`

Even with the 10k Optimizer enabled: `OR` conditions cost less than their equivalent `AND` conditions.

Proof of Concept

- Compare in Remix this example contract's 2 diffs (or any test-contract of your choice, as experimentation always show the same results):

```
pragma solidity 0.8.13;

contract Test {
    bool isOpen;
    bool channelPreviouslyOpen;

    function boolTest() external view returns (uint) {
-       if (isOpen && !channelPreviouslyOpen) {
+       if (!(!isOpen || channelPreviouslyOpen)) {
            return 1;
-       } else if (!isOpen && channelPreviouslyOpen) {
+       } else if (!(isOpen || !channelPreviouslyOpen)) {
            return 2;
        }
    }

    function setBools(bool _isOpen, bool _channelPreviouslyOpen)
        isOpen = _isOpen;
        channelPreviouslyOpen= _channelPreviouslyOpen;
    }
}
```

- Notice that, even with the 10k Optimizer, the red diff version costs **8719 gas**, while the green diff version costs **8707 gas**, effectively **saving 12 gas**.

Affected Code

Added together, it's possible to save a significant amount of gas by replacing the `&&` conditions by their `||` equivalent in the solution.

- `ConduitController.sol#updateChannel()`

Use `!(!isOpen || channelPreviouslyOpen)` instead of `isOpen && !channelPreviouslyOpen` and use `!(isOpen || !channelPreviouslyOpen)` instead of `!isOpen && channelPreviouslyOpen`:

```
File: ConduitController.sol
- 141:         if (isOpen && !channelPreviouslyOpen) {
+ 141:         if (!(!isOpen || channelPreviouslyOpen))
...
- 149:         } else if (!isOpen && channelPreviouslyOpen) {
```

```
+ 149:         } else if (!(isOpen || !channelPreviouslyOpen))
```

- `OrderValidator.sol#_validateOrderAndUpdateStatus()`

Use `!(!(numerator < denominator) || !_doesNotSupportPartialFills(orderParameters.orderType))` instead of `numerator < denominator && _doesNotSupportPartialFills(orderParameters.orderType:`

```
contracts/lib/OrderValidator.sol:
  142           if (
- 143:              numerator < denominator &&
- 144               _doesNotSupportPartialFills(orderParameters.
+ 143:              !(!(numerator < denominator) ||
+ 144               !_doesNotSupportPartialFills(orderParameters
  145           ) {
```

- `OrderValidator.sol#_cancel()`

Use `!(msg.sender == offerer || msg.sender == zone)` instead of `msg.sender != offerer && msg.sender != zone` here:

```
- 280:                 if (msg.sender != offerer && msg.sender
+ 280:                 if (!(msg.sender == offerer || msg.sende
```

- `SignatureVerification.sol#_assertValidSignature()`

Use `!(v == 27 || v == 28)` instead of `v != 27 && v != 28:`

```
contracts/lib/SignatureVerification.sol:
- 78:              if (v != 27 && v != 28) {
+ 78:              if (!(v == 27 || v == 28))
```

- `ZoneInteraction.sol#_assertRestrictedBasicOrderValidity()`

Use `!(!(uint256(orderType) > 1) || msg.sender == zone || msg.sender == offerer)` instead of `uint256(orderType) > 1 && msg.sender != zone && msg.sender != offerer`:

```
contracts/lib/ZoneInteraction.sol:
   46          if (
-   47:             uint256(orderType) > 1 &&
-   48:             msg.sender != zone &&
-   49              msg.sender != offerer
+   47:             !(!(uint256(orderType) > 1) ||
+   48:             msg.sender == zone ||
+   49              msg.sender == offerer)
   50          ) {
```

- `ZoneInteraction.sol#_assertRestrictedAdvancedOrderValidity()` (1)

Use `!(!(uint256(orderType) > 1) || msg.sender == zone || msg.sender == offerer)` instead of `uint256(orderType) > 1 && msg.sender != zone && msg.sender != offerer`:

```
  115          if (
-  116:             uint256(orderType) > 1 &&
-  117:             msg.sender != zone &&
-  118              msg.sender != offerer
+  116:             !(!(uint256(orderType) > 1) ||
+  117:             msg.sender == zone ||
+  118              msg.sender == offerer)
  119          ) {
```

- `ZoneInteraction.sol#_assertRestrictedAdvancedOrderValidity()` (2)

Use `!(advancedOrder.extraData.length != 0 || criteriaResolvers.length != 0)` instead of `advancedOrder.extraData.length == 0 && criteriaResolvers.length == 0`:

```
  121              if (
-  122:                 advancedOrder.extraData.length == 0 &&
-  123                  criteriaResolvers.length == 0
```

```
+  122:                 !(advancedOrder.extraData.length != 0 ||
+  123                  criteriaResolvers.length != 0)
   124              ) {
```

[G-11] Bytes constants are more efficient than string constants

From the Solidity doc:

> If you can limit the length to a certain number of bytes, always use one of bytes1 to bytes32 because they are much cheaper.

Why do Solidity examples use bytes32 type instead of string?

> bytes32 uses less gas because it fits in a single word of the EVM, and string is a dynamically sized-type which has current limitations in Solidity (such as can't be returned from a function to a contract).

If data can fit into 32 bytes, then you should use bytes32 datatype rather than bytes or strings as it is cheaper in solidity. Basically, any fixed size variable in solidity is cheaper than variable size. That will save gas on the contract.

Instances of `string constant` that can be replaced by `bytes(1..32) constant` :

```
reference/lib/ReferenceConsiderationBase.sol:
  29:     string internal constant _NAME = "Consideration";
  30:     string internal constant _VERSION = "rc.1";
```

[G-12] An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop consumes more gas than necessary.

In the best case scenario (length read on a memory variable), caching the array length in the stack saves around 3 gas per iteration. In the worst case scenario (external calls at each iteration), the amount of gas wasted can be massive.

Here, lengths are only read from memory.

Consider storing the array's length in a variable before the for-loop, and use this newly created variable instead:

```
lib/OrderCombiner.sol:247:                for (uint256 j = 0; j
lib/OrderCombiner.sol:291:                for (uint256 j = 0; j
lib/OrderCombiner.sol:598:                for (uint256 j = 0; j
lib/OrderCombiner.sol:621:            for (uint256 i = 0; i < execut
lib/OrderFulfiller.sol:217:                for (uint256 i = 0; i < o
lib/OrderFulfiller.sol:306:                for (uint256 i = 0; i < o
```

[G-13] Increments can be unchecked

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Affected code

```
lib/CriteriaResolution.sol:56:                for (uint256 i = 0; i
lib/CriteriaResolution.sol:166:                for (uint256 i = 0; i
lib/CriteriaResolution.sol:184:                    for (uint256 j =
lib/CriteriaResolution.sol:199:                    for (uint256 j =
lib/OrderCombiner.sol:181:                for (uint256 i = 0; i < to
lib/OrderCombiner.sol:247:                    for (uint256 j = 0; j
lib/OrderCombiner.sol:291:                    for (uint256 j = 0; j
lib/OrderCombiner.sol:373:                for (uint256 i = 0; i < to
lib/OrderCombiner.sol:473:                for (uint256 i = 0; i < to
lib/OrderCombiner.sol:498:                for (uint256 i = 0; i < to
lib/OrderCombiner.sol:577:                for (uint256 i = 0; i < to
lib/OrderCombiner.sol:598:                    for (uint256 j = 0; j
lib/OrderCombiner.sol:754:                for (uint256 i = 0; i < to
lib/OrderFulfiller.sol:471:                for (uint256 i = 0; i < t
```

The code would go from:

```
for (uint256 i; i < numIterations; ++i) {
 // ...
}
```

to:

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

The risk of overflow is inexistant for `uint256` here.

Note that this is already applied at some places in the solution. As an example:

```
contracts/conduit/Conduit.sol:
   66:         for (uint256 i = 0; i < totalStandardTransfers; )
   ...
   74:             unchecked {
   75                 ++i;
   76             }

  130:         for (uint256 i = 0; i < totalStandardTransfers; )
  ...
  138:             unchecked {
  139                 ++i;
  140             }

contracts/lib/BasicOrderFulfiller.sol:
   948:         for (uint256 i = 0; i < totalAdditionalRecipient
   ...
   975:             unchecked {
   976                 ++i;
   977             }


  1040:         for (uint256 i = 0; i < totalAdditionalRecipient
  ...
  1064:             unchecked {
  1065                 ++i;
```

```
1066                 }
```

[G-14] No need to explicitly initialize variables with default values

This finding is only true without the Optimizer

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Affected code:

```
conduit/Conduit.sol:66:         for (uint256 i = 0; i < totalStar
conduit/Conduit.sol:130:         for (uint256 i = 0; i < totalSta
lib/AmountDeriver.sol:44:             uint256 extraCeiling = 0;
lib/BasicOrderFulfiller.sol:948:            for (uint256 i = 0; i <
lib/BasicOrderFulfiller.sol:1040:            for (uint256 i = 0; i <
lib/CriteriaResolution.sol:56:              for (uint256 i = 0; i
lib/CriteriaResolution.sol:166:              for (uint256 i = 0; i
lib/CriteriaResolution.sol:184:                  for (uint256 j =
lib/CriteriaResolution.sol:199:                  for (uint256 j =
lib/OrderCombiner.sol:181:              for (uint256 i = 0; i < to
lib/OrderCombiner.sol:247:                  for (uint256 j = 0; j
lib/OrderCombiner.sol:291:                  for (uint256 j = 0; j
lib/OrderCombiner.sol:373:              for (uint256 i = 0; i < to
lib/OrderCombiner.sol:470:              uint256 totalFilteredExecu
lib/OrderCombiner.sol:473:              for (uint256 i = 0; i < to
lib/OrderCombiner.sol:498:              for (uint256 i = 0; i < to
lib/OrderCombiner.sol:577:              for (uint256 i = 0; i < to
lib/OrderCombiner.sol:598:                  for (uint256 j = 0; j
lib/OrderCombiner.sol:621:          for (uint256 i = 0; i < execut
lib/OrderCombiner.sol:751:              uint256 totalFilteredExecu
lib/OrderCombiner.sol:754:              for (uint256 i = 0; i < to
lib/OrderFulfiller.sol:217:              for (uint256 i = 0; i < o
lib/OrderFulfiller.sol:306:              for (uint256 i = 0; i < o
lib/OrderFulfiller.sol:471:              for (uint256 i = 0; i < t
lib/OrderValidator.sol:272:              for (uint256 i = 0; i < t
```

```
lib/OrderValidator.sol:350:            for (uint256 i = 0; i < t
```

I suggest removing explicit initializations for default values.

[G-15] `abi.encode()` is less efficient than `abi.encodePacked()`

Changing `abi.encode` function to `abi.encodePacked` can save gas since the `abi.encode` function pads extra null bytes at the end of the call data, which is unnecessary. Also, in general, `abi.encodePacked` is more gas-efficient (see Solidity-Encode-Gas-Comparison).

Consider using `abi.encodePacked()` here:

```
contracts/lib/ConsiderationBase.sol:
  77          return keccak256(
  78:             abi.encode(
  79                  _EIP_712_DOMAIN_TYPEHASH,
  80                  _NAME_HASH,
  81                  _VERSION_HASH,
  82                  block.chainid,
  83                  address(this)
  84              )
  85          );
```

Consider using the assembly equivalent for `abi.encodePacked()` here:

```
reference/lib/ReferenceBasicOrderFulfiller.sol:
  513           orderHash = keccak256(
  514:              abi.encode(
  515                   hashes.typeHash,
  516                   parameters.offerer,
  517                   parameters.zone,
  518                   hashes.offerItemsHash,
  519                   hashes.receivedItemsHash,
  520                   fulfillmentItemTypes.orderType,
  521                   parameters.startTime,
  522                   parameters.endTime,
  523                   parameters.zoneHash,
```

```
524                 parameters.salt,
525                 parameters.offererConduitKey,
526                 nonce
527             )
528         );

609             hashes.considerationHashes[0] = keccak256(
610:                abi.encode(
611                     hashes.typeHash,
612                     primaryConsiderationItem.itemType,
613                     primaryConsiderationItem.token,
614                     primaryConsiderationItem.identifierOr
615                     primaryConsiderationItem.startAmount,
616                     primaryConsiderationItem.endAmount,
617                     primaryConsiderationItem.recipient
618                 )
619             );

684                 hashes.considerationHashes[recipientCount
685:                    abi.encode(
686                         hashes.typeHash,
687                         additionalRecipientItem.itemType,
688                         additionalRecipientItem.token,
689                         additionalRecipientItem.identifie
690                         additionalRecipientItem.startAmou
691                         additionalRecipientItem.endAmount
692                         additionalRecipientItem.recipient
693                     )
694                 );
695             }

756                 keccak256(
757:                    abi.encode(
758                         hashes.typeHash,
759                         offerItem.itemType,
760                         offerItem.token,
761                         offerItem.identifier,
762                         offerItem.amount,
763                         offerItem.amount //Assembly uses
764                     )
765                 )
```

reference/lib/ReferenceConsiderationBase.sol:

```
117         return keccak256(
118:            abi.encode(
119                 _eip712DomainTypeHash,
```

```
  120                 _nameHash,
  121                 _versionHash,
  122                 block.chainid,
  123                 address(this)
  124             )
  125         );

reference/lib/ReferenceGettersAndDerivers.sol:
   41         return
   42             keccak256(
   43:                abi.encode(
   44                     _OFFER_ITEM_TYPEHASH,
   45                     offerItem.itemType,
   46                     offerItem.token,
   47                     offerItem.identifierOrCriteria,
   48                     offerItem.startAmount,
   49                     offerItem.endAmount
   50                 )
   51             );
   52     }

   66         return
   67             keccak256(
   68:                abi.encode(
   69                     _CONSIDERATION_ITEM_TYPEHASH,
   70                     considerationItem.itemType,
   71                     considerationItem.token,
   72                     considerationItem.identifierOrCriteri
   73                     considerationItem.startAmount,
   74                     considerationItem.endAmount,
   75                     considerationItem.recipient
   76                 )
   77             );

  123         return
  124             keccak256(
  125:                abi.encode(
  126                     _ORDER_TYPEHASH,
  127                     orderParameters.offerer,
  128                     orderParameters.zone,
  129                     keccak256(abi.encodePacked(offerHashe
  130                     keccak256(abi.encodePacked(considerat
  131                     orderParameters.orderType,
  132                     orderParameters.startTime,
  133                     orderParameters.endTime,
  134                     orderParameters.zoneHash,
```

```
135                     orderParameters.salt,
136                     orderParameters.conduitKey,
137                     nonce
138                 )
139             );
```

Notice that this is already used at other places for a similar situation:

```
reference/lib/ReferenceBasicOrderFulfiller.sol:
  769             hashes.offerItemsHash = keccak256(
  770                 abi.encodePacked(offerItemHashes)
  771             );

reference/lib/ReferenceGettersAndDerivers.sol:
  129                     keccak256(abi.encodePacked(offerHashe
  130                     keccak256(abi.encodePacked(considerat
```

HardlyDifficult (judge) commented:

> **[G-01] Cheap Contract Deployment Through Clones**
>
> Deploying clones would save cost when Conduits are created, however it also increases the cost to use the conduits created. That increase has a tiny impact, but I assume it was intentional to favor the end-users here - creating conduits will be relatively rare and reserved for platforms and/or power users.
>
> **[G-02] ConduitController.sol#createConduit(): Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it**
>
> This general tactic can often result in significant savings, but I ran the recommended change and here it only saves ~100 gas on `createConduit`.
>
> **[G-03] ConduitController.sol#acceptOwnership(): Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it**
>
> This will save some gas, but `acceptOwnership` is not a critical code path so an optimization here would not impact many transactions.

[G-04] `OrderValidator.sol#_validateBasicOrderAndUpdateStatus()`: **Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it**

This is similar to the recommendation in issue **#60** (although #60 appears to be a bit more thorough) and provides fairly significant savings on critical code paths.

[G-05] `OrderValidator.sol#_validateBasicOrderAndUpdateStatus()`: **avoid an unnecessary SSTORE by not writing a default value**

This is a safe change to make since `_verifyOrderStatus` will first revert if the order is already canceled. Considered independently from the rec above, this saves ~300 gas on `fulfillBasicOrder`.

[G-06] OrderCombiner.sol: ++totalFilteredExecutions costs less gas compared to totalFilteredExecutions += 1
[G-07] OrderCombiner.sol: —maximumFulfilled costs less gas compared to maximumFulfilled—
[G-08] `FulfillmentApplier.sol#_applyFulfillment()`: **Unchecking arithmetics operations that can't underflow/overflow**
[G-09] /reference: Unchecking arithmetics operations that can't underflow/overflow
[G-12] An array's length should be cached to save gas in for-loops
[G-13] Increments can be unchecked

Yes these should provide some savings.

[G-10] `OR` **conditions cost less than their equivalent** `AND` **conditions ("NOT(something is false)" costs less than "everything is true")**

It's unfortunate that the optimizer cannot handle scenarios like this automatically... There does appear to be a small win here, but it's debatable whether the impact to readability is worth it here.

[G-11] Bytes constants are more efficient than string constants

These changes seem to be focused on getters, it's not clear it would impact gas for any transactions.

> **[G-14] No need to explicitly initialize variables with default values**

> This appears to be handled by the optimizer automatically now. Testing did not change the gas results.

> **[G-15] abi.encode() is less efficient than abi.encodePacked()**

> This recommendation causes tests to fail, suggesting this change violates the EIP-712 standard.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top