TESSERA

Tessera - Versus contest Findings & Analysis Report

2023-04-18

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 Tessera smart contract system written in Solidity. The audit contest took place between December 16—December 19 2022.

Following the C4 audit contest, 3 wardens (IllIllI, Lambda, and gzeon) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit contest report.

Wardens

5 Wardens contributed reports to the Tessera - Versus contest:

1. IllIllI

2. Lambda
3. Trust
4. cccz
5. gzeon

This contest was judged by hickuphh3.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 18 unique vulnerabilities. Of these vulnerabilities, 9 received a risk rating in the category of HIGH severity and 9 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 4 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 2 reports recommending gas optimizations.

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

Scope

The code under review can be found within the C4 Tessera - Versus contest repository, and is composed of 5 smart contracts written in the Solidity programming language and includes 795 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (9)

[H-01] GroupBuy does not check return value of call

Submitted by Lambda*, also found by* Trust *and* IllIllI

https://github.com/code-423n4/2022-12-tessera/blob/1e408ebc1c4fdcc72678ea7f21a94d38855ccc0b/src/modules/GroupBuy.sol#L265

https://github.com/code-423n4/2022-12-tessera/blob/1e408ebc1c4fdcc72678ea7f21a94d38855ccc0b/src/modules/GroupBuy.sol#L283

Impact

Both usages of `call` do not check if the transfer of ETH was succesful:

```
    payable(msg.sender).call{value: contribution}("");
    ...
    payable(msg.sender).call{value: balance}("");
```

This can become very problematic when the recipient is a smart contract that reverts (for instance, temporarily) in its `receive` function. Then, `GroupBuy` still assumes that this ETH was transferred out and sets the balance to 0 or deletes `userContributions[_poolId][msg.sender]`, although no ETH was transferred. This leads to a loss of funds for the recipient.

Proof Of Concept

We assume that the recipient is a smart contract that performs some logic in its `receive` function. For instance, it can be a nice feature for some people to automatically convert all incoming ETH into another token using an AMM. However, it can happen that the used AMM has too little liquidity at the moment or the slippage of a swap would be too high, leading to a revert in the receing contract. In such a scenario, the `GroupBuy` contract still thinks that the call was succesful, leading to lost funds for the recipient.

Recommended Mitigation Steps

`require` that the call was successful.

HickupHH3 (judge) commented:

> Keeping as High severity because of valid use case and resulting loss of funds if the receiving contract reverts, but the tx doesn't.

stevennevins (Tessera) confirmed and mitigated:

> https://github.com/fractional-company/modular-fractional/pull/204

> **Status:** Mitigation confirmed by gzeon, IllIllI, and Lambda.

[H-02] GroupBuy: Lost ETH when the NFT is bought for less than the minimum reserve price

Submitted by Lambda, *also found by* gzeon

The `purchase` function does not require that an NFT is bought for exactly `minReservePrices[_poolId] * filledQuantities[_poolId]`, the price is only not allowed to be greater:

```
    if (_price > minReservePrices[_poolId] * filledQuantities[_poolI
                revert InvalidPurchase();
```

This makes sense because it is not sensible to pay more when the purchase also succeeds with a smaller amount. However, the logic within `claim` does assume that the NFT was bought for `minReservePrices[_poolId]`. It decreases from `contribution` the quantity times the reserve price for all bids:

```
    contribution -= quantity * reservePrice;
```

Only the remaining amount is reimbursed to the user, which can lead to a loss of funds.

Proof Of Concept

Let's say that `filledQuantities[_poolId] = 100` and `minReservePrices[_poolId]` (i.e., the lowest bid) was 1 ETH. However, it was possible to buy the NFT for only 50 ETH. When a user has contributed 20 * 1 ETH, he does not get anything back when calling `claim`, although only 10 ETH (0.5 ETH * 20) of his contributions were used to buy the NFT. The overall loss of funds for all contributors is 50 ETH.

Recommended Mitigation Steps

Set `minReservePrices[_poolId]` to `_price / filledQuantities[_poolId]` after a purchase.

stevennevins (Tessera) disagreed with severity and commented:

> Not sure I agree with the severity. The mechanism is essentially users pre-state their X interest at Y quantity and so a user can never "pay" at a price greater than they essentially agreed to. We will look into ways to better handle the change and as it related to #19. I would mark this as Medium.

mehtaculous (Tessera) confirmed

HickupHH3 (judge) commented:

> Funds are considered lost if the NFT was bought at a discounted price, and cannot be recovered, right? Would keep at High severity if it's the case.

stevennevins (Tessera) commented:

> Yeah correct, confirmed.

stevennevins (Tessera) mitigated:

> https://github.com/fractional-company/modular-fractional/pull/207

Status: Mitigation not confirmed. Full details in reports from gzeon and IllIllI. Also included in Mitigation Review section below.

[H-03] Groupbuy: `_verifyUnsuccessfulState` and `_verifySuccessfulState` both can return true when block.timestamp == pool.terminationPeriod

Submitted by Lambda

https://github.com/code-423n4/2022-12-tessera/blob/1e408ebc1c4fdcc72678ea7f21a94d38855ccc0b/src/modules/GroupBuy.sol#L455

https://github.com/code-423n4/2022-12-tessera/blob/1e408ebc1c4fdcc72678ea7f21a94d38855ccc0b/src/modules/GroupBuy.sol#L478

Impact

The functions `_verifyUnsuccessfulState` and `_verifySuccessfulState` should always have a differing behavior with regards to reversion, i.e. when one does not revert, the other should revert. In one condition, this is not true. Namely, when we have `pool.success == false` and `block.timestamp == pool.terminationPeriod`, this check within `_verifyUnsuccessfulState` is `false`:

```
if (pool.success || block.timestamp > pool.terminationPeriod) re
```

Similarly, this check within `_verifySuccessfulState` is also `false`:

```
    if (!pool.success && block.timestamp < pool.terminationPeriod) r
```

Because this breaks a fundamental invariant of the contract, there are probably multiple ways to exploit it. One way an attacker can exploit is by calling `claim` (to get his contribution back completely), bidding again with a higher value than his previous contributions (to get his contributions back again).

Proof Of Concept

Let's assume we are at timestamp `pool.terminationPeriod`. Attacker Charlie has performed the lowest bid with quantity 10 and price 1 ETH. He calls `claim` to get his 10 ETH back. Now, he calls `contribute` with a quantity of 10 and a price of 2 ETH. Because this bid is higher than his previous one (which was the lowest one), his `pendingBalances` is set to 10 ETH (for the deleted entries) and his `userContributions` is set to 20 ETH (for this new contribution). He can now call `claim` again to get back his 20 ETH in `userContributions`, but also the 10 ETH in `pendingBalances`. Like that, he has stolen 10 ETH (and could use this attack pattern to drain the whole contract).

Recommended Mitigation Steps

Change `<` in `_verifySuccessfulState` to `<=`.

[HickupHH3 (judge) commented]:

> Given that block timestamp period for ETH mainnet is now a constant 12s, the probability of a block timestamp being equal to `terminationPeriod` is 1/12 (~8.3%), which is non-trivial.

[stevennevins (Tessera) confirmed and mitigated]:

> https://github.com/fractional-company/modular-fractional/pull/203

> **Status:** Mitigation confirmed by gzeon, IllIllI, and Lambda.

[H-04] OptimisticListingSeaport.propose sets

pendingBalances of newly added proposer instead of previous one

Submitted by Lambda, also found by Trust

In `OptimisticListingSeaport.propose`, `pendingBalances` is set to the collateral. The purpose of this is that the proposer of a previous proposal can withdraw his collateral afterwards. However, this is done on the storage variable `proposedListing` after the new listing is already set:

```
_setListing(proposedListing, msg.sender, _collateral, _pricePerT

// Sets collateral amount to pending balances for withdrawal
pendingBalances[_vault][proposedListing.proposer] += proposedLis
```

Because of that, it will actually set `pendingBalances` of the new proposer. Therefore, the old proposer loses his collateral and the new one can make proposals for free.

Proof Of Concept

```
--- a/test/seaport/OptimisticListingSeaport.t.sol
+++ b/test/seaport/OptimisticListingSeaport.t.sol
@@ -379,8 +379,11 @@ contract OptimisticListingSeaportTest is Se
     /// ===== LIST =====
     /// ================
     function testList(uint256 _collateral, uint256 _price) publ
         // setup
         testPropose(_collateral, _price);
+        assertEq(optimistic.pendingBalances(vault, bob), 0);
         _increaseTime(PROPOSAL_PERIOD);
         _collateral = _boundCollateral(_collateral, bobTokenBal
         _price = _boundPrice(_price);
```

This test fails and `optimistic.pendingBalances(vault, bob)` is equal to `_collateral`.

Recommended Mitigation Steps

Run `pendingBalances[_vault][proposedListing.proposer] += proposedListing.collateral;` before the `_setListing` call, in which case the above PoC no longer works.

HickupHH3 (judge) commented:

> Because of that, it will actually set pendingBalances of the new proposer. Therefore, the old proposer loses his collateral and the new one can make proposals for free.

> Seems like intended behaviour to me (actually set pendingBalances of the new proposer). The old proposer wouldn't be losing his collateral because his pendingBalances would've been set when he called `propose()`.

mehtaculous (Tessera) confirmed and commented:

> Agree with severity. The suggested solution makes sense.

stevennevins (Tessera) mitigated:

> https://github.com/fractional-company/modular-fractional/pull/202

> **Status:** Mitigation confirmed by gzeon, IllIllI, and Lambda.

[H-05] Attacker can DOS OptimisticListing with very low cost

Submitted by gzeon, also found by Trust, Trust, and cccz

The only check on a new proposal is that it is priced lower than the existing proposal. It does not constrain on the `_collateral` supplied (except it will revert in `\_verifyBalance` if set to 0). Anyone can block normal proposal creation by creating a proposal with lower price but `\_collateral == 1`. When a high total supply is used, the price of each Rae is negligible and enables an attacker to DOS the protocol.

This violated the `prevent a user from holding a vault hostage and never letting the piece be reasonably bought` requirement.

Proof of Concept

For any proposal, an attacker can deny it with `\_collateral = 1` and `\_price = price - 1`.

If he does not want the NFT to be sold, he can reject the proposal himself, resetting the contract state.

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L112-L116

```
        // Reverts if price per token is not lower than both the
        if (
            _pricePerToken >= proposedListing.pricePerToken ||
            _pricePerToken >= activeListings[_vault].pricePerTok
        ) revert NotLower();
```

Add this test to OptimisticListingSeaport.t.sol:

```
    function testProposeRevertLowerTotalValue() public {
        uint256 _collateral = 100;
        uint256 _price = 100;
        // setup
        testPropose(_collateral, _price);
        lowerPrice = pricePerToken - 1;
        // execute
        vm.expectRevert();
        _propose(eve, vault, 1, lowerPrice, offer);
        // expect
        _assertListing(eve, 1, lowerPrice, block.timestamp);
        _assertTokenBalance(eve, token, tokenId, eveTokenBalance
    }
```

[FAIL. Reason: Call did not revert as expected]

Tools Used

Foundry

Recommended Mitigation Steps

Require the total value of the new collateral to be greater than the previous.

This however still allows a Rae holder with sufficiently large holding to block proposal by creating a new proposal and immediately reject it himself.

stevennevins (Tessera) confirmed

HickupHH3 (judge) commented:

> Best report for Foundry POC + the following statement:

> This violated the `prevent a user from holding a vault hostage and never letting the piece be reasonably bought` requirement.

[H-06] Funds are permanently stuck in OptimisticListingSeaport.sol contract if active proposal is executed after new proposal is pending.

Submitted by Trust

`\_constructOrder` is called in `propose()`, OptimisticListingSeaport.sol. It fills the order params stored in proposedListings[_vault].

```
    {
        orderParams.offerer = _vault;
        orderParams.startTime = block.timestamp;
        // order doesn't expire in human time scales and needs expli
        orderParams.endTime = type(uint256).max;
        orderParams.zone = zone;
        // 0: no partial fills, anyone can execute
        orderParams.orderType = OrderType.FULL_OPEN;
        orderParams.conduitKey = conduitKey;
        // 1 Consideration for the listing itself + 1 consideration
        orderParams.totalOriginalConsiderationItems = 3;
    }
```

Importantly, it updates the order hash associated with the vault:

```
vaultOrderHash[_vault] = _getOrderHash(orderParams, counter);
```

There is only one other use of `vaultOrderHash`, in `\_verifySale()`.

```
    function _verifySale(address _vault) internal view returns (bool
        (bool isValidated, bool isCancelled, uint256 totalFilled, ui
            seaport
        ).getOrderStatus(vaultOrderHash[_vault]);
        if (isValidated && !isCancelled && totalFilled > 0 && totalF
            status = true;
        }
    }
```

This function gets order information from the order hash, and makes sure the order is completely fulfilled.

After NFT sell has completed, `cash()` is used to distribute income ETH:

```
    function cash(address _vault, bytes32[] calldata _burnProof) ext
        // Reverts if vault is not registered
        (address token, uint256 id) = _verifyVault(_vault);
        // Reverts if active listing has not been settled
        Listing storage activeListing = activeListings[_vault];
        // Reverts if listing has not been sold
              // -------------- _verifySale MUST BE TRUE ---------
        if (!_verifySale(_vault)) {
            revert NotSold();
        } else if (activeListing.collateral != 0) {
            uint256 collateral = activeListing.collateral;
            activeListing.collateral = 0;
            // Sets collateral amount to pending balances for withdr
            pendingBalances[_vault][activeListing.proposer] = collat
        }
```

As long as sale is not complete, `cash()` can't be called as highlighted. The issue is that `vaultOrderHash[_vault]` is not protected during the lifetime of an active proposal. If another proposal is proposed and then the sell using active proposal takes place, `cash()` will keep reverting. Funds are stuck in listing contract.

We can try to be clever and call `propose()` again with the same parameters to create an identical orderID, which will make `vaultOrderHash[_vault]` fine again and allow `cash()` to go through. But order params contain block.timestamp which will certainly be different which will make the hash different.

Impact

Funds are permanently stuck in OptimisticListingSeaport.sol contract if active proposal is executed after new proposal is pending.

Proof of Concept

1. User A calls `propose()`, setting proposedListing. vaultOrderHash=X
2. PROPOSAL_PERIOD passes , list is called promoting the listing to activeListing.
3. Another user, malicious or innocent, proposes another proposal. vaultOrderHash=Y
4. Sell goes down due to OpenSea validation confirmed on activeListing.
5. `\_verifySale` will never return true because we can never got vaultOrderHash to be X
6. cash() is bricked. Money is stuck in contract.

Recommended Mitigation Steps

Keep the order hash in the Listing structure rather than a single one per vault.

mehtaculous (Tessera) confirmed and commented:

> Agree with High severity. Solution is to move `orderHash` to Listing struct so that active and proposed listings can have separate order hashes.

stevennevins (Tessera) mitigated:

> https://github.com/fractional-company/modular-fractional/pull/211

> **Status:** Mitigation confirmed by gzeon, IllIllI, and Lambda.

[H-07] User loses collateral converted to pendingBalance when `cash()` or `list()` is called

Submitted by Trust, also found by Lambda

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L295

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L232

Description

In OptimisticListingOpensea, there are several functions which update pendingBalances of a proposer:

1. `list()`
2. `cash()`
3. `propose()`

Unfortunately, in `list()` and `cash()` the = operator is used instead of += when writing the new pendingBalances. For example:

```
function cash(address _vault, bytes32[] calldata _burnProof) ext
    // Reverts if vault is not registered
    (address token, uint256 id) = _verifyVault(_vault);
    // Reverts if active listing has not been settled
    Listing storage activeListing = activeListings[_vault];
    // Reverts if listing has not been sold
    if (!_verifySale(_vault)) {
        revert NotSold();
    } else if (activeListing.collateral != 0) {
        uint256 collateral = activeListing.collateral;
        activeListing.collateral = 0;
        // Sets collateral amount to pending balances for withdr
        pendingBalances[_vault][activeListing.proposer] = collat
    }
        ...
```

pendingBalances is not guaranteed to be zero. There could be funds from previous proposals which are not yet collected. Propose updates pendingBalance correctly:

```
    // Sets collateral amount to pending balances for withdrawal
    pendingBalances[_vault][proposedListing.proposer] += proposedLis
```

So, when propose is followed by another propose(), the pendingBalance is updated correctly, but in cash and list we don't account for pre-existing balance. This issue would manifest even after the fix suggested in the issue "User can send a proposal and instantly take back their collateral" because reject functions would increment the pendingBalance and then it would be overriden.

Impact

User loses collateral converted to pendingBalance when `cash()` or `list()` is called.

Proof of Concept

1. User calls `propose()` and gets pendingBalance = x
2. User calls `propose()` with an improved proposal and gets pendingBalance = 1.5x
3. proposal is successfull and the listing purchased the NFT
4. `cash()` is called to convert the Raes to ETH amount from the sell. pendingBalance is overridden by the current "collateral" value. pendingBalance = 0.5x
5. User loses x collateral value which is stuck in the contract

Recommended Mitigation Steps

Change the = operator to += in `list()` and `cash()`.

[mehtaculous (Tessera) confirmed and commented](#):

> Agree with High severity. Solution is to replace `=` with `+=` for both `list` and `cash` functions.

[stevennevins (Tessera) mitigated](#):

> https://github.com/fractional-company/modular-fractional/pull/202

> **Status:** Mitigation confirmed by gzeon, IllIllI, and Lambda.

[H-08] Attacker can steal the amount collected so far in the GroupBuy for NFT purchase.

Submitted by Trust, also found by IllIllI and Lambda

`purchase()` in GroupBuy.sol executes the purchase call for the group. After safety checks, the NFT is bought with `\_market`'s `execute()` function. Supposedly it deploys a vault which owns the NFT. The code makes sure the vault is the new owner of the NFT and exits.

```
    // Executes purchase order transaction through market buyer cont
    address vault = IMarketBuyer(_market).execute{value: _price}(_pu
    // Checks if NFT contract supports ERC165 and interface ID of EF
    if (ERC165Checker.supportsInterface(_nftContract, _INTERFACE_ID_
        // Verifes vault is owner of ERC-721 token
        if (IERC721(_nftContract).ownerOf(_tokenId) != vault) revert
    } else {
        // Verifies vault is owner of CryptoPunk token
        if (ICryptoPunk(_nftContract).punkIndexToAddress(_tokenId) !
            revert UnsuccessfulPurchase();
    }

    // Stores mapping value of poolId to newly deployed vault
    poolToVault[_poolId] = vault;
    // Sets pool state to successful
    poolInfo[_poolId].success = true;
    // Emits event for purchasing NFT at given price
    emit Purchase(_poolId, vault, _nftContract, _tokenId, _price);
```

The issue is that `\_market` user-supplied variable is not validated at all. Attacker can pass their malicious contract, which uses the passed funds to buy the NFT and store it in attacker's wallet. It will return the NFT-holding wallet so the checks will pass. As a result, attacker has the NFT while they could have contributed nothing to the GroupBuy. Attacker can also just steal the supplied ETH and return the current address which holds the NFT.

Impact

Attacker can steal the amount collected so far in the GroupBuy for NFT purchase.

Proof of Concept

1. Group assembles and raises funds to buy NFT X
2. Attacker calls `purchase()` and supplies their malicious contract in `\_market`, as described.
3. Attacker receives raised funds totalling `minReservePrices[_poolId] * filledQuantities[_poolId]`, as checked in line 182.

Recommended Mitigation Steps

`\_market` should be whitelisted, or supplied in createPool stage and able to be scrutinized.

mehtaculous (Tessera) confirmed and commented:

> Agree with High severity. Solution is to check that the `vault` deployed from the MarketBuyer is actually registered through the `VaultRegistry`. This would confirm that the vault is not a user address

stevennevins (Tessera) mitigated:

> https://github.com/fractional-company/modular-fractional/pull/201

Status: Mitigation not confirmed. Full details in report from gzeon, and also included in the Mitigation Review section below.

[H-09] GroupBuy can be drained of all ETH.

Submitted by Trust, also found by Lambda

`purchase()` in GroupBuy faciilitates the purchasing of an NFT after enough contributions were gathered. Another report titled *"Attacker can steal the amount collected so far in the GroupBuy for NFT purchase*" describes a high impact bug in purchase. It is advised to read that first for context.

Additionally, `purchase()` is vulnerable to a re-entrancy exploit which can be *chained* or *not chained* to the `\_market` issue to steal *the entire* ETH stored in GroupBuy, rather than being capped to `minReservePrices[_poolId] * filledQuantities[_poolId]`.

Attacker may take control of execution using this call:

```
    // Executes purchase order transaction through market buyer cont
    address vault = IMarketBuyer(_market).execute{value: _price}(_pu
```

It could occur either by exploiting the unvalidated `\_market` vulnerability , or by abusing an existing market that uses a user address in `\_purchaseOrder`.

There is no re-entrancy protection in `purchase()` call:

```
    function purchase(
        uint256 _poolId,
        address _market,
        address _nftContract,
        uint256 _tokenId,
        uint256 _price,
        bytes memory _purchaseOrder,
        bytes32[] memory _purchaseProof
    ) external {
```

`\_verifyUnsuccessfulState()` needs to not revert for purchase call. It checks the pool.success flag: `if (pool.success || block.timestamp > pool.terminationPeriod) revert InvalidState();`

However, success is only set as the last thing in `purchase()`:

```
        // Stores mapping value of poolId to newly deployed vault
        poolToVault[_poolId] = vault;
        // Sets pool state to successful
        poolInfo[_poolId].success = true;
        // Emits event for purchasing NFT at given price
        emit Purchase(_poolId, vault, _nftContract, _tokenId, _price
    }
```

Therefore, attacker can re-enter purchase() function multiple times, each time extracting the maximum allowed price. If attacker uses the controlled `\_market` exploit, the function will return the current NFT owner, so when all the functions unwind they will keep setting success to true and exit nicely.

Impact

GroupBuy can be drained of all ETH.

Proof of Concept

1. GroupBuy holds 1500 ETH, from various bids
2. maximum allowed price (`minReservePrices[_poolId] * filledQuantities[_poolId]`) is 50 * 20 = 1000 ETH
3. purchase(1000 ETH) is called
    1. GroupBuy sends attacker 1000 ETH and calls `execute()`
        1. `execute()` calls purchase(500ETH)
            1. GroupBuy sends attacker 500 ETH and calls `execute()`
                1. execute returns NFT owner address

            2. GroupBuy sees returned address is NFT owner. Marks success and returns
        2. execute returns NFT owner address
    2. GroupBuy sees returned address is NFT owner. Marks success and returns
4. Attacker is left with 1500 ETH. Previous exploit alone can only net 1000ETH. Additionally, this exploit can be chained to any trusted MarketBuyer which passes control to user for purchasing and storing in vault, and then returns a valid vault.

Recommended Mitigation Steps

Add a re-entrancy guard to `purchase()` function. Also, change success variable before performing external contract calls.

mehtaculous (Tessera) confirmed and commented:

> Agree with High severity. Instead of adding `re-entrancy` tag to `purchase` function, pool state simply needs to be updated to `success` before execution.

> In regards to:

> or by abusing an existing market that uses a user address in _purchaseOrder.

> This is not considered an issue since users will most likely NOT contribute to a pool where they are not familiar with the NFT and / or contract. Since the NFT contract is set when the pool is created, it should not matter whether the contract is malicious or is for an existing market that uses a user address, the pool will just be disregarded.

stevennevins (Tessera) mitigated:

> https://github.com/fractional-company/modular-fractional/pull/201

> **Status:** Mitigation confirmed by gzeon, IllIllI, and Lambda.

Medium Risk Findings (9)

[M-01] GroupBuy may purchase NFT not in the allowed list

Submitted by gzeon, also found by Trust and Lambda

When `_purchaseProof.length == 0`, GroupBuy.purchase compare the tokenId with the merkleRoot. This allows any tokenId that matches the merkleRoot to be purchased, even if they are not included in the allow list during setup.

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L186-L188

```
        if (_purchaseProof.length == 0) {
            // Hashes tokenId to verify merkle root if proof is
            if (bytes32(_tokenId) != merkleRoot) revert InvalidP
```

Proof of Concept

Add the following to GroupBuy.t.sol. It would still revert (since no such nft existed) but not as expected.

```
    // modified from testPurchaseRevertInvalidProof
    function testPurchaseRevertInvalidTokenIdZeroLength() public
        // setup
        testContributeSuccess();
        // exploit
        uint256 invalidPunkId = uint256(nftMerkleRoot);
        bytes32[] memory invalidPurchaseProof = new bytes32[](0)
        // expect
        vm.expectRevert(INVALID_PROOF_ERROR);
        // execute
        _purchase(
            address(this),
            currentId,
            address(punksBuyer),
            address(punks),
            invalidPunkId,
            minValue,
            purchaseOrder,
            invalidPurchaseProof
            );
        }
```

Tools Used

Foundry

Recommended Mitigation Steps

Hash the tokenId even if there is only when length is 1.

stevennevins (Tessera) disagreed with severity

HickupHH3 (judge) decreased severity to Medium and commented:

> Best report because Foundry POC.

[M-02] Attacker can delay proposal rejection

Submitted by gzeon

In `OptimisticListingSeaport.rejectProposal`, it reverts if `proposedListing.collateral < _amount`. An attacker can therefore monitor the mempool, reducing the `proposedListing.collateral` to `_amount - 1` by frontrunning the `rejectProposal` call and delaying the rejection. The attacker may even be able to deny the rejection when the deadline passes.

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L145

```
            if (proposedListing.collateral < _amount) revert Insuffi
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L153

```
            proposedListing.collateral -= _amount;
```

Proof of Concept

1. Attacker proposes a 10000 collateral at a very low price
2. Bob tries to reject it by purchasing the 10000 collateral
3. Attacker sees Bob's tx in the mempool, frontruns it to reject 1 unit
4. The proposedListing.collateral is now 9999
5. Bob's call reverted
6. This keeps happening until PROPOSAL_PERIOD passes or Bob gave up because of gas paid on failing tx
7. Attacker buys the NFT at a very low price

Recommended Mitigation Steps

When `proposedListing.collateral < _amount`, set `\_amount` to proposedListing.collateral and refund the excess.

stevennevins (Tessera) disagreed with severity and commented:

> While annoying, I think this is a Medium severity issue. The attacker has to under price their proposal, defend this under priced proposal from other users and front run each purchase > 1, while selling their Raes at a loss, and it would be relatively costly for griefer to defend their underpriced proposal for the duration of the proposal period. Users could purchase 1 Rae at a time without risk of front running.

HickupHH3 (judge) decreased severity to Medium and commented:

> Agree with Medium severity given the external requirements needed to pull this off.

[M-03] Users that send funds at a price lower than the current low bid have the funds locked

Submitted by IllIllI, also found by Lambda and cccz

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L114-L150

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L301-L303

Vulnerability details

If a user contributes funds after there is no more supply left, and they don't provide a price higher than the current minimum bid, they will be unable to withdraw their funds while the NFT remains unbought.

Impact

Ether becomes stuck until and unless the NFT is bought, which may never happen.

Proof of Concept

When making a contribution, the user calls the `payable contribute()` function. If the supply has already been filled (`fillAtAnyPriceQuantity` is zero), the bid isn't inserted into the queue, so the new bid is not tracked anywhere. When the function reaches `processBidsInQueue()` ...:

```
// File: src/modules/GroupBuy.sol : GroupBuy.contribute()   #1

99          function contribute(
100             uint256 _poolId,
101             uint256 _quantity,
102             uint256 _price
103 @>      ) public payable {
104             // Reverts if pool ID is not valid
105             _verifyPool(_poolId);
106             // Reverts if NFT has already been purchased OR t
107             (, uint48 totalSupply, , , ) = _verifyUnsuccessfu
108             // Reverts if ether contribution amount per Rae i
109             if (msg.value < _quantity * minBidPrices[_poolId]
110                 revert InvalidContribution();
```

```
            // Reverts if ether payment amount is not equal t
            if (msg.value != _quantity * _price) revert Inval

            // Updates user and pool contribution amounts
            userContributions[_poolId][msg.sender] += msg.val
            totalContributions[_poolId] += msg.value;

            // Calculates remaining supply based on total pos
            uint256 remainingSupply = totalSupply - filledQua
            // Calculates quantity amount being filled at any
            uint256 fillAtAnyPriceQuantity = remainingSupply

            // Checks if quantity amount being filled is grea
@>          if (fillAtAnyPriceQuantity > 0) {
                // Inserts bid into end of queue
                bidPriorityQueues[_poolId].insert(msg.sender,
                // Increments total amount of filled quantiti
                filledQuantities[_poolId] += fillAtAnyPriceQu
            }

            // Calculates unfilled quantity amount based on c
            uint256 unfilledQuantity = _quantity - fillAtAnyP
            // Processes bids in queue to recalculate unfille
@>          unfilledQuantity = processBidsInQueue(_poolId, un

            // Recalculates filled quantity amount based on u
            uint256 filledQuantity = _quantity - unfilledQuar
            // Updates minimum reserve price if filled quanti
            if (filledQuantity > 0) minReservePrices[_poolId]

            // Emits event for contributing ether to pool bas
            emit Contribute(
                _poolId,
                msg.sender,
                msg.value,
                _quantity,
                _price,
                minReservePrices[_poolId]
            );
        }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L99-L150

...if the price isn't higher than the lowest bid, the while loop is broken out of, with `pendingBalances` having never been updated, and the function does not revert:

```
// File: src/modules/GroupBuy.sol : GroupBuy.processBidsInQueue(

291         function processBidsInQueue(
292             uint256 _poolId,
293             uint256 _quantity,
294             uint256 _price
295         ) private returns (uint256 quantity) {
296             quantity = _quantity;
297             while (quantity > 0) {
298                 // Retrieves lowest bid in queue
299                 Bid storage lowestBid = bidPriorityQueues[_pc
300                 // Breaks out of while loop if given price is
301 @>              if (_price < lowestBid.price) {
302 @>                  break;
303 @>              }
304
305                 uint256 lowestBidQuantity = lowestBid.quantit
306                 // Checks if lowest bid quantity amount is gr
307                 if (lowestBidQuantity > quantity) {
308                     // Decrements given quantity amount from
309                     lowestBid.quantity -= quantity;
310                     // Calculates partial contribution of bic
311                     uint256 contribution = quantity * lowestE
312
313:                    // Decrements partial contribution amount
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L291-L313

In order for a user to get funds back, the amount must have been stored in `pendingBalances`, and since this is never done, all funds contributed during the `contribute()` call become property of the `GroupBuy` contract, with the user being unable to withdraw...:

```
// File: src/modules/GroupBuy.sol : GroupBuy.withdrawBalance()

274         function withdrawBalance() public {
```

```
275             // Reverts if caller balance is insufficient
276 @>          uint256 balance = pendingBalances[msg.sender];
277 @>          if (balance == 0) revert InsufficientBalance();
278
279             // Resets pending balance amount
280             delete pendingBalances[msg.sender];
281
282             // Transfers pending ether balance to caller
283             payable(msg.sender).call{value: balance}("");
284:        }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L274-L284

...until the order has gone through, and they can `claim()` excess funds, but there likely won't be any, due to the separate MEV bug I raised:

```
// File: src/modules/GroupBuy.sol : GroupBuy.contribution   #4

228         function claim(uint256 _poolId, bytes32[] calldata _n
229             // Reverts if pool ID is not valid
230             _verifyPool(_poolId);
231             // Reverts if purchase has not been made AND tern
232             (, , , bool success, ) = _verifySuccessfulState(_
233             // Reverts if contribution balance of user is ins
234 @>          uint256 contribution = userContributions[_poolId]
235             if (contribution == 0) revert InsufficientBalance
236
237             // Deletes user contribution from storage
238             delete userContributions[_poolId][msg.sender];
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L228-L244

Recommended Mitigation Steps

`revert()` if the price is lower than the min bid, and the queue is already full stevennevins (Tessera) confirmed and mitigated:

> https://github.com/fractional-company/modular-fractional/pull/206

> **Status:** Mitigation confirmed by gzeon and Lambda.

[M-04] Priority queue min accounting breaks when nodes are split in two

Submitted by IllIllI, also found by Trust

The README states `If two users place bids at the same price but with different quantities, the queue will pull from the bid with a higher quantity first`, but the data-structure used for implementing this logic, is not used properly and essentially has its data corrupted when a large bid that is the current minimum bid, is split into two parts, so that a more favorable price can be used for a fraction of the large bid. The underlying issue is that one of the tree nodes is modified, without re-shuffling that node's location in the tree.

Impact

The minimum bid as told by the priority queue will be wrong, leading to the wrong bids being allowed to withdraw their funds, and being kicked out of the fraction of bids that are used to buy the NFT.

Proof of Concept

The priority queue using a binary tree within an array to efficiently navigate and find the current minimum based on a node and its children. The sorting of the nodes in the tree is based, in part, on the quantity in the case where two bids have the same price:

```
// File: src/lib/MinPriorityQueue.sol : MinPriorityQueue.isGreat

111          function isGreater(
112              Queue storage self,
113              uint256 i,
114              uint256 j
```

```
115         ) private view returns (bool) {
116             Bid memory bidI = self.bidIdToBidMap[self.bidIdLi
117             Bid memory bidJ = self.bidIdToBidMap[self.bidIdLi
118 @>          if (bidI.price == bidJ.price) {
119 @>              return bidI.quantity <= bidJ.quantity;
120             }
121             return bidI.price > bidJ.price;
122:        }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/lib/MinPriorityQueue.sol#L111-L122

The algorithm of the binary tree only works when the nodes are properly sorted. The sorting is corrupted when a node is modified, without removing it from the tree and re-inserting it:

```
// File: src/modules/GroupBuy.sol : GroupBuy.processBidsInQueue(

299                 Bid storage lowestBid = bidPriorityQueues[_po
300                 // Breaks out of while loop if given price is
301                 if (_price < lowestBid.price) {
302                     break;
303                 }
304
305                 uint256 lowestBidQuantity = lowestBid.quantit
306                 // Checks if lowest bid quantity amount is gr
307                 if (lowestBidQuantity > quantity) {
308                     // Decrements given quantity amount from
309 @>                  lowestBid.quantity -= quantity;
310                     // Calculates partial contribution of bic
311                     uint256 contribution = quantity * lowestE
312
313                     // Decrements partial contribution amount
314                     totalContributions[_poolId] -= contributi
315                     userContributions[_poolId][lowestBid.owne
316                     // Increments pending balance of lowest k
317                     pendingBalances[lowestBid.owner] += contr
318
319:                    // Inserts new bid with given quantity an
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L299-L319

Let’s say that the tree looks like this:

```
                A:(p:100,q:10)
                /             \
           B:(p:100,q:10)   C:(<whatever>)
           /            \
    D:(whatever)    E:(whatever)
```

If A is modified so that q (quantity) goes from 10 to 5, B should now be at the root of the tree, since it has the larger size, and would be considered the smaller node. When another node is added, say, `F:(p:100,q:6)` , the algorithm will see that F has a larger size than A, and so A will be popped out as the min, even though B should have been. All nodes that are under B (which may be a lot of the nodes if they all entered at the same price/quantity) essentially become invisible under various scenarios, which means the users that own those bids will not be able to withdraw their funds, even if they really are the lowest bid that deserves to be pushed out of the queue. Note that the swimming up that is done for `F` will not re-shuffle `B` since, according to the algorithm, `F` will start as a child of `C` , and `B` is not in the list of parent nodes of `C` .

Recommended Mitigation Steps

When modifying nodes of the tree, remove them first, then re-add them after modification.

stevennevins (Tessera) confirmed and mitigated:

> https://github.com/fractional-company/modular-fractional/pull/212

> **Status:** Mitigation confirmed by gzeon and Lambda.

[M-05] Orders may not be fillable due to missing approvals

Submitted by IIIIIII, also found by gzeon

Not all `IERC20` implementations `revert()` when there's a failure in `approve()`. If one of these tokens returns false, there is no check for whether this has happened during the order listing validation, so it will only be detected when the order is attempted.

Impact

If the approval failure isn't detected, the listing will never be fillable, because the funds won't be able to be pulled from the opensea conduit. Once this happens, and if it's detected, the only way to fix it is to create a counter-listing at a lower price (which may be below the market value of the tokens), waiting for the order to expire (which it may never), or by buying out all of the Rae to cancel the order (very expensive and defeats the purpose of pooling funds in the first place).

Proof of Concept

The return value of `approve()` isn't checked, so the order will be allowed to be listed without having approved the conduit:

```
// File: src/seaport/targets/SeaportLister.sol : SeaportLister.v

                for (uint256 i; i < ordersLength; ++i) {
                    uint256 offerLength = _orders[i].parameter
                    for (uint256 j; j < offerLength; ++j) {
                        OfferItem memory offer = _orders[i].pa
                        address token = offer.token;
                        ItemType itemType = offer.itemType;
                        if (itemType == ItemType.ERC721)
                            IERC721(token).setApprovalForAll(c
                        if (itemType == ItemType.ERC1155)
                            IERC1155(token).setApprovalForAll(
                        if (itemType == ItemType.ERC20)
@>                          IERC20(token).approve(conduit, typ
                    }
                }
            }
            // Validates the order on-chain so no signature is
            assert(ConsiderationInterface(_consideration).vali
        }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/targets/SeaportLister.sol#L29-L46

Recommended Mitigation Steps

Use OpenZeppelin's `safeApprove()`, which checks the return code and reverts if it's not success.

mehtaculous (Tessera) disputed and commented:

> Disagree with validity. The listing would just need to be canceled and a new order would be created (without the ERC20 token that is not able to be approved)

HickupHH3 (judge) commented:

> `cancel()` can only be performed by the proposer, or through `rejectActive()`:

> or by buying out all of the Rae to cancel the order (very expensive and defeats the purpose of pooling funds in the first place).

> While unlikely, it is an attack vector to hold user funds hostage.

stevennevins (Tessera) acknowledged

[M-06] Only one `GroupBuy` can ever use USDT or similar tokens with front-running approval protections

Submitted by IllIllI

Calling `approve()` without first calling `approve(0)` if the current approval is non-zero will revert with some tokens, such as Tether (USDT). While Tether is known to do this, it applies to other tokens as well, which are trying to protect against this attack vector.

Impact

Only the first listing will start with the conduit's approval at zero and will be able to change it to the maximum. Thereafter, every attempt to approve that same token will revert, causing any order using this lister to revert, including a re-listing at a lower price, which the protocol allows for.

Proof of Concept

The seaport conduit address is set in the constructor as an immutable variable, so it's not possible to change it once the issue is hit:

```
// File: src/seaport/targets/SeaportLister.sol : SeaportLister.c

19          constructor(address _conduit) {
20 @>           conduit = _conduit;
21:         }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/targets/SeaportLister.sol#L19-L21

The approval is set to the maximum without checking whether it's already the maximum:

```
// File: src/seaport/targets/SeaportLister.sol : SeaportLister.v

29                  for (uint256 i; i < ordersLength; ++i) {
30                      uint256 offerLength = _orders[i].parameter
31                      for (uint256 j; j < offerLength; ++j) {
32                          OfferItem memory offer = _orders[i].pa
33                          address token = offer.token;
34                          ItemType itemType = offer.itemType;
35                          if (itemType == ItemType.ERC721)
36                              IERC721(token).setApprovalForAll(c
37                          if (itemType == ItemType.ERC1155)
38                              IERC1155(token).setApprovalForAll(
39                          if (itemType == ItemType.ERC20)
40 @>                           IERC20(token).approve(conduit, typ
41                      }
42                  }
43              }
44              // Validates the order on-chain so no signature is
```

```
        assert(ConsiderationInterface(_consideration).vali
    }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/targets/SeaportLister.sol#L29-L46

The README states: `The Tessera Protocol is designed around the concept of Hyperstructures, which are crypto protocols that can run for free and forever, without maintenance, interruption or intermediaries`, and having to deploy a new `SeaportLister` in order to have a fresh conduit that also needs to be deployed, is not `without maintenance or interruption`.

Recommended Mitigation Steps

Always reset the approval to zero before changing it to the maximum.

mehtaculous (Tessera) disputed and commented:

> Disagree with approval issue since the conduit will not be able to front-run the approvals.

> However, the issue regarding the conduit as an immutable variable is valid and it seems that a solution would be to pass the conduit in as a parameter for every `validateListing` call.

HickupHH3 (judge) commented:

> The issue here isn't about the frontrunning attack vector, but about not being able to reset the approval back to the maximum because some token implementations require the allowance can only be set to non-zero from zero, ie. non-zero -> non-zero is disallowed.

> Meaning, once the allowance is partially used, subsequent attempts to approve back to `type(uint256).max` will fail.

stevennevins (Tessera) commented:

> I think this title is mislabeled and they meant SeaportOL. The Vault delegate calls to the Seaport Lister so these approvals would be scoped to the Vault so multiple listing are possible through the target contract. But a vault that relists wouldn't be able to relist an ERC20 with front running protection.

[M-07] Loss of ETH for proposer when it is a contract that doesn't have fallback function.

Submitted by [Trust]

`sendEthOrWeth()` is used in several locations in OptimisticListingSeaport:

1. rejectProposal - sent to proposer
2. rejectActive - sent to proposer
3. cash - sent to msg.sender

This is the implementation of sendEthOrWeth:

```
    function _attemptETHTransfer(address _to, uint256 _value) interr
        // Here increase the gas limit a reasonable amount above the
        // to send ETH to the recipient.
        // NOTE: This might allow the recipient to attempt a limitec
        (success, ) = _to.call{value: _value, gas: 30000}("");
    }
    /// @notice Sends eth or weth to an address
    /// @param _to Address to send to
    /// @param _value Amount to send
    function _sendEthOrWeth(address _to, uint256 _value) internal {
        if (!_attemptETHTransfer(_to, _value)) {
            WETH(WETH_ADDRESS).deposit{value: _value}();
            WETH(WETH_ADDRESS).transfer(_to, _value);
        }
    }
```

The issue is that the receive could be a contract that does not have a fallback function. In this scenario, _attemptETHTransfer will fail and WETH would be transferred to the contract. It is likely that it bricks those funds for the contract as there is no reason it would support interaction with WETH tokens.

It can be reasonably assumed that developers will develop contracts which will interact with OptimisticListingSeaport using proposals. They are not warned and are likely to suffer losses.

Impact

Loss of ETH for proposer when it is a contract that doesn't have fallback function.

Recommended Mitigation Steps

Either enforce that proposer is an EOA or take in a recipient address for ETH transfers.

HickupHH3 (judge) commented:

> The argument here is about the contract being able to handle ETH but not WETH. If the ETH transfer fails (eg. gas used exceeds the 30k sent), then funds would be stuck.

> On the fence regarding severity here.

stevennevins (Tessera) confirmed, but disagreed with severity and commented:

> I actually more agree with this being an issue:

> The argument here is about the contract being able to handle ETH but not WETH. If the ETH transfer fails (eg. gas used exceeds the 30k sent), then funds would be stuck.

> But it's not clear to me that is what was originally highlighted in the description of the issue.

HickupHH3 (judge) commented:

> Yeah it's not fully clear because the premise is the contract not having a fallback function, but the intended effect of not being able to handle WETH is.

> It is likely that it bricks those funds for the contract as there is no reason it would support interaction with WETH tokens.

[M-08] Earlier bidders get cut out of future NFT holdings by bidders specifying the same price.

Submitted by Trust, also found by cccz

In GroupBuy module, users can call contribute to get a piece of the NFT pie. There are two stages in transforming the msg.value to holdings in the NFT.

1. filling at any price(supply is not yet saturated)

```
uint256 fillAtAnyPriceQuantity = remainingSupply < _quantity ? r
// Checks if quantity amount being filled is greater than 0
if (fillAtAnyPriceQuantity > 0) {
    // Inserts bid into end of queue
    bidPriorityQueues[_poolId].insert(msg.sender, _price, fillAt
    // Increments total amount of filled quantities
    filledQuantities[_poolId] += fillAtAnyPriceQuantity;
}
```

2. Trim out lower price offers to make room for current higher offer.

```
// Calculates unfilled quantity amount based on desired quantity
uint256 unfilledQuantity = _quantity - fillAtAnyPriceQuantity;
// Processes bids in queue to recalculate unfilled quantity amou
unfilledQuantity = processBidsInQueue(_poolId, unfilledQuantity,
```

The while loop in `processBidsInQueue` will keep removing existing bids with lower price and create new queue entries for currently processed bid. When it reached a bid with a higher price than msg.sender's price, it will break:

```
while (quantity > 0) {
    // Retrieves lowest bid in queue
    Bid storage lowestBid = bidPriorityQueues[_poolId].getMin();
    // Breaks out of while loop if given price is less than thar
    if (_price < lowestBid.price) {
        break;
    }
```

The issue is that when `_price == lowestBid.price`, we don't break and current bid will kick out older bid, as can be seen here:

```
    // Decrements given quantity amount from lowest bid quantity
    lowestBid.quantity -= quantity;
    // Calculates partial contribution of bid by quantity amount anc
    uint256 contribution = quantity * lowestBid.price;
    // Decrements partial contribution amount of lowest bid from tot
    totalContributions[_poolId] -= contribution;
    userContributions[_poolId][lowestBid.owner] -= contribution;
    // Increments pending balance of lowest bid owner
    pendingBalances[lowestBid.owner] += contribution;
    // Inserts new bid with given quantity amount into proper positi
    bidPriorityQueues[_poolId].insert(msg.sender, _price, quantity);
```

The described behavior goes against what the docs describe will happen when two equal priced bids collide.

Impact

Earlier bidders get cut out of future NFT holdings by bidders specifying the same price.

Recommended Mitigation Steps

Change the < to <= in the if condition:

```
    if (_price <= lowestBid.price) {
        break;
    }
```

IllIIII (warden) commented:

> @HickupHH3 @Trust the final code block in this submission comes from here, which falls under the case where the price is the same and the quantity is different. The readme states `If two users place bids at the same price but with different quantities, the queue will pull from the bid with`

> `a higher quantity first`, and as the submission shows, it's pulling from the higher quantity first - doesn't that mean this finding is invalid?

Trust (warden) commented:

> I agree the example shown is accidentally the wrong block (if clause) and the misbehavior occurs in the else block:
>
> ```
>             } else {
>                 // Calculates total contribution of bid by quant
>                 uint256 contribution = lowestBid.quantity * lowe
>
>                 // Decrements full contribution amount of lowest
>                 totalContributions[_poolId] -= contribution;
>                 userContributions[_poolId][lowestBid.owner] -= c
>                 // Increments pending balance of lowest bid owne
>                 pendingBalances[lowestBid.owner] += contributior
>
>                 // Removes lowest bid in queue
>                 bidPriorityQueues[_poolId].delMin();
>                 // Inserts new bid with lowest bid quantity amou
>                 bidPriorityQueues[_poolId].insert(msg.sender, _p
>                 // Decrements lowest bid quantity from total qua
>                 quantity -= lowestBidQuantity;
> ```
>
> The issue described still manifests in this block.

stevennevins (Tessera) confirmed and mitigated

> https://github.com/fractional-company/modular-fractional/pull/212

> **Status:** Mitigation confirmed by gzeon and Lambda.

[M-09] GroupBuys that are completely filled still don't raise stated target amount

Submitted by Trust*, also found by* gzeon `createPool()` in GroupBuy.sol creates a new contribution pool around an NFT. It specifies a target `\_initialPrice` as minimum amount of ETH the NFT will cost, and `\_totalSupply` which is the number of Raes to be minted on purchase success.

minBidPrices is calculated from the two numbers. All future bids must be at least minBidPrices. It is assumed that if the totalSupply of Raes is filled up, the group will collect the initialPrice.

```
    // Calculates minimum bid price based on initial price of NFT ar
    minBidPrices[currentId] = _initialPrice / _totalSupply;
```

The issue is that division rounding error will make minBidPrices too low. Therefore, when all Raes are minted using minBidPrices price: `minBidPrices[currentId] * _totalSupply != _initialPrice`

Therefore, not enough money has been collected to fund the purchase.

It can be assumed that most people will use minBidPrices to drive the price they will choose. Therefore, even after discovering that the Group has not raised enough after filling the supply pool, it will be very hard to get everyone to top up the contribution by a bit. This is because the settled price which is collected from all contributions is minReservePrices, which is always the minimum price deposited.

Code in contribute that updates minReservePrices:

```
    // Updates minimum reserve price if filled quantity amount is gr
    if (filledQuantity > 0) minReservePrices[_poolId] = getMinPrice(
```

The check in purchase() that we don't charge more than minReservePrices from each contribution:

```
    if (_price > minReservePrices[_poolId] * filledQuantities[_poolI
        revert InvalidPurchase();
```

We can see an important contract functionality is not working as expected which will impair NFT purchases.

Impact

GroupBuys that are completely filled still don't raise stated target amount.

Recommended Mitigation Steps

Round the minBidPrices up, rather than down. It will ensure enough funds are collected.

HickupHH3 (judge) commented:

> Agree that the division rounding down will be a problem. Reasonable for a RAE holder to create a new pool where the `_initialPrice` isn't exactly divisible by `_totalSupply`:

> Eg. `_initialPrice = 100`, _totalSupply = 8 `. Then purchase will always revert with` InvalidPurchase()`because:

- `minBidPrices[currentId] = 100 / 8` and
- `_price (100) > minReservePrices[_poolId] * filledQuantities[_poolId]) = 12 * 8 = 96` is true.

stevennevins (Tessera) confirmed and mitigated:

> https://github.com/fractional-company/modular-fractional/pull/213

> **Status:** Mitigation confirmed by gzeon, IllIllI, and Lambda.

Low Risk and Non-Critical Issues

For this contest, 4 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by IllIllI received the top score from the judge.

The following wardens also submitted reports: gzeon, cccz, and Lambda .

Low Risk Issues Summary

	Issue	Instances
[L-01]	Bid size is an unfair ordering metric	1
[L-02]	Users may DOS themselves with a lot of smalle payments	1
[L-03]	Empty `receive()` / `payable fallback()` function does not authorize requests	2
[L-04]	`require()` should be used instead of `assert()`	2
[L-05]	Missing checks for `address(0x0)` when assigning values to `address` state variables	12

Total: 18 instances over 5 issues

[L-01] Bid size is an unfair ordering metric

The README states that this is intentional, so I've filed it as Low rather than Medium, but giving priority to bids with the smaller quantity is not a fair ordering mechanic. A person with a lot of funds may have gotten that by pooling externally to the contract, and it's not fair to kick them out of the pool earlier than another address that came in later.

There is 1 instance of this issue:

```
File: /src/lib/MinPriorityQueue.sol

111       function isGreater(
112           Queue storage self,
113           uint256 i,
114           uint256 j
115       ) private view returns (bool) {
116           Bid memory bidI = self.bidIdToBidMap[self.bidIdList
117           Bid memory bidJ = self.bidIdToBidMap[self.bidIdList
118           if (bidI.price == bidJ.price) {
119               return bidI.quantity <= bidJ.quantity;
120           }
121           return bidI.price > bidJ.price;
122:     }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/lib/MinPriorityQueue.sol#L111-L122

[L-02] Users may DOS themselves with a lot of smalle payments

If a user has to contribute to a pool via lots of dust payments (e.g. if they only have enough money each week to spend a few wei), they may eventually add enough payments that when it's time to claim their excess, their for-loop below exceeds the block gas limit.

There is 1 instance of this issue:

```
File: /src/modules/GroupBuy.sol

247            if (success) {
248                for (uint256 i; i < length; ++i) {
249                    // Gets bid quantity from storage
250:                   Bid storage bid = bidPriorityQueues[_poolI
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L247-L250

[L-03] Empty `receive()` / `payable fallback()` function does not authorize requests

If the intention is for the Ether to be used, the function should call another function, otherwise it should revert (e.g. `require(msg.sender == address(weth))`). Having no access control on the function means that someone may send Ether to the contract, and have no way to get anything back out, which is a loss of funds. If the concern is having to spend a small amount of gas to check the sender against an immutable address, the code should at least have a function to rescue unused Ether.

There are 2 instances of this issue:

```
File: src/punks/protoforms/PunksMarketBuyer.sol

41:        receive() external payable {}
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/punks/protoforms/PunksMarketBuyer.sol#L41

```
File: src/seaport/modules/OptimisticListingSeaport.sol

83:        receive() external payable {}
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L83

[L-04] `require()` should be used instead of `assert()`

Prior to solidity version 0.8.0, hitting an assert consumes the **remainder of the transaction's available gas** rather than returning it, as `require()` / `revert()` do. `assert()` should be avoided even past solidity version 0.8.0 as its documentation states that "The assert function creates an error of type Panic(uint256). ... Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix".

There are 2 instances of this issue:

```
File: src/seaport/targets/SeaportLister.sol

45:            assert(ConsiderationInterface(_consideration).vali

52:            assert(ConsiderationInterface(_consideration).canc
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/targets/SeaportLister.sol#L45

[L-05] Missing checks for `address(0x0)` when assigning values to `address` state variables

There are 12 instances of this issue:

```
File: src/punks/protoforms/PunksMarketBuyer.sol

32:          registry = _registry;

33:          wrapper = _wrapper;

34:          listing = _listing;
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/punks/protoforms/PunksMarketBuyer.sol#L32

```
File: src/seaport/modules/OptimisticListingSeaport.sol

71:          registry = _registry;

72:          seaport = _seaport;

73:          zone = _zone;

75:          supply = _supply;

76:          seaportLister = _seaportLister;

77:          feeReceiver = _feeReceiver;

78:          OPENSEA_RECIPIENT = _openseaRecipient;

338:         feeReceiver = _new;
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L71

```
File: src/seaport/targets/SeaportLister.sol

20:            conduit = _conduit;
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/targets/SeaportLister.sol#L20

Non-Critical Issues Summary

	Issue	Instances
[N-01]	Debugging functions should be moved to a child class rather than being deployed	1
[N-02]	Typos	4
[N-03]	`public` functions not called by the contract should be declared `external` instead	10
[N-04]	`constant` s should be defined rather than using magic numbers	4
[N-05]	Missing event and or timelock for critical parameter change	1
[N-06]	NatSpec is incomplete	7
[N-07]	Consider using `delete` rather than assigning zero to clear values	4
[N-08]	Contracts should have full test coverage	1
[N-09]	Large or complicated code bases should implement fuzzing tests	1

Total: 33 instances over 9 issues

[N-01] Debugging functions should be moved to a child class rather than being deployed

There is 1 instance of this issue:

```
File: /src/modules/GroupBuy.sol

402      function printQueue(uint256 _poolId) public view {
403          uint256 counter;
404          uint256 index = 1;
405          MinPriorityQueue.Queue storage queue = bidPriority(
406          uint256 numBids = queue.numBids;
407          while (counter < numBids) {
408              Bid memory bid = queue.bidIdToBidMap[index];
409              if (bid.bidId == 0) {
410                  ++index;
411                  continue;
412              }
413              ++index;
414              ++counter;
415          }
416:     }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L402-L416

[N-02] Typos

There are 4 instances of this issue:

```
File: src/lib/MinPriorityQueue.sol

/// @audit addreses
22:           ///@notice map addreses to bids they own
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/lib/MinPriorityQueue.sol#L22

```
File: src/modules/GroupBuy.sol

/// @audit equalt
179:          // Reverts if NFT contract is not equalt to NFT cc

/// @audit Verifes
208:              // Verifes vault is owner of ERC-721 token

/// @audit specifc
286:      /// @notice Attempts to accept bid for specifc quantit
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L179

[N-03] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

There are 10 instances of this issue:

```
File: src/lib/MinPriorityQueue.sol

27:      function initialize(Queue storage self) public {

36:      function getNumBids(Queue storage self) public view re

41:      function getMin(Queue storage self) public view returr

90:      function delMin(Queue storage self) public returns (Bi
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/lib/MinPriorityQueue.sol#L27

```
File: src/modules/GroupBuy.sol
```

```
346        function getBidInQueue(uint256 _poolId, uint256 _bidI
347            public
348            view
349            returns (
350                uint256 bidId,
351                address owner,
352                uint256 price,
353:               uint256 quantity

371:       function getNextBidId(uint256 _poolId) public view ret

377:       function getNumBids(uint256 _poolId) public view retur

384:       function getBidQuantity(uint256 _poolId, uint256 _bidI

402:       function printQueue(uint256 _poolId) public view {
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L346-L353

```
File: src/seaport/modules/OptimisticListingSeaport.sol

218:       function list(address _vault, bytes32[] calldata _list
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L218

[N-04] `constant`s should be defined rather than using magic numbers

Even [assembly](https://github.com/code-423n4/2022-05-opensea-seaport/blob/9d7ce4d08bf3c3010304a0476a785c70c0e90ae7/contracts/lib/TokenTransferrer.sol#L35-L39) can benefit from using readable constants instead of hex/numeric literals.

There are 4 instances of this issue:

```
File: src/seaport/modules/OptimisticListingSeaport.sol
```

```
/// @audit 3
350:            permissions = new Permission[](3);

/// @audit 3
385:                orderParams.totalOriginalConsiderationItems =

/// @audit 40
395:            uint256 openseaFees = _listingPrice / 40;

/// @audit 20
396:            uint256 tesseraFees = _listingPrice / 20;
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L350

[N-05] Missing event and or timelock for critical parameter change

Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

There is 1 instance of this issue:

```
File: src/seaport/modules/OptimisticListingSeaport.sol

336        function updateFeeReceiver(address payable _new) exter
337            if (msg.sender != feeReceiver) revert NotAuthorize
338            feeReceiver = _new;
339:       }
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L336-L339

[N-06] NatSpec is incomplete

There are 7 instances of this issue:

```
File: src/modules/GroupBuy.sol

/// @audit Missing: '@return'
          /// @param _poolId ID of the pool
          /// @param _bidId ID of the bid in queue
          function getBidInQueue(uint256 _poolId, uint256 _bidI
              public
              view
              returns (
                  uint256 bidId,
                  address owner,
                  uint256 price,
                  uint256 quantity

/// @audit Missing: '@return'
          /// @notice Gets minimum bid price of queue for given
          /// @param _poolId ID of the pool
          function getMinPrice(uint256 _poolId) public view retu

/// @audit Missing: '@return'
          /// @notice Gets next bidId in queue of given pool
          /// @param _poolId ID of the pool
          function getNextBidId(uint256 _poolId) public view ret

/// @audit Missing: '@return'
          /// @notice Gets total number of bids in queue for gi
          /// @param _poolId ID of the pool
          function getNumBids(uint256 _poolId) public view retur

/// @audit Missing: '@return'
          /// @param _poolId ID of the pool
          /// @param _bidId ID of the bid in queue
          function getBidQuantity(uint256 _poolId, uint256 _bid

/// @audit Missing: '@return'
          /// @param _poolId ID of the pool
          /// @param _owner Address of the owner
          function getOwnerToBidIds(uint256 _poolId, address _o
              public
              view
              returns (uint256[] memory)
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/Group

Buy.sol#L344-L353

```
File: src/punks/protoforms/PunksMarketBuyer.sol

/// @audit Missing: '@return'
44          /// @param _order Bytes value of the necessary order p
45          /// return vault Address of the deployed vault
46:         function execute(bytes memory _order) external payable
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/punks/protoforms/PunksMarketBuyer.sol#L44-L46

[N-07] Consider using `delete` rather than assigning zero to clear values

The `delete` keyword more closely matches the semantics of what is being done, and draws more attention to the changing of state, which may lead to a more thorough audit of its associated logic.

There are 4 instances of this issue:

```
File: src/modules/GroupBuy.sol

253:                    bid.quantity = 0;
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/modules/GroupBuy.sol#L253

```
File: src/seaport/modules/OptimisticListingSeaport.sol

293:                activeListing.collateral = 0;

325:            pendingBalances[_vault][_to] = 0;

437:            orderParams.totalOriginalConsiderationItems = 0;
```

https://github.com/code-423n4/2022-12-tessera/blob/f37a11407da2af844bbfe868e1422e3665a5f8e4/src/seaport/modules/OptimisticListingSeaport.sol#L293

[N-08] Contracts should have full test coverage

While 100% code coverage does not guarantee that there are no bugs, it often will catch easy-to-find bugs, and will ensure that there are fewer regressions when the code invariably has to be modified. Furthermore, in order to get full coverage, code authors will often have to re-organize their code so that it is more modular, so that each component can be tested separately, which reduces interdependencies between modules and layers, and makes for code that is easier to reason about and audit.

There is 1 instance of this issue:

```
File: src/lib/MinPriorityQueue.sol
```

[N-09] Large or complicated code bases should implement fuzzing tests

Large code bases, or code with lots of inline-assembly, complicated math, or complicated interactions between multiple contracts, should implement fuzzing tests. Fuzzers such as Echidna require the test writer to come up with invariants which should not be violated under any circumstances, and the fuzzer tests various inputs and function calls to ensure that the invariants always hold. Even code with 100% code coverage can still have bugs due to the order of the operations a user performs, and fuzzers, with properly and extensively-written invariants, can close this testing gap significantly.

There is 1 instance of this issue:

```
File: src/lib/MinPriorityQueue.sol
```

HickupHH3 (judge) commented:

> Tough one between this and #4 as both contain very useful findings. While I like that #4 has more context specific findings, I decided to go with this one because of the number of findings made and its comprehensiveness.

stevennevins (Tessera) confirmed and mitigated:

> https://github.com/fractional-company/modular-fractional/pull/205

Gas Optimizations

For this contest, 2 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **gzeon** received the top score from the judge.

The following wardens also submitted reports: IllIllI.

Gas Optimizations Summary

Disclaimer: report extended from 4naly3er tool

	Issue	Instances
[G-01]	Use assembly to check for `address(0)`	2
[G-02]	Cache array length outside of loop	2
[G-03]	State variables should be cached in stack variables rather than re-reading them from storage	5
[G-04]	Use calldata instead of memory for function arguments that do not get mutated	5
[G-05]	Use Custom Errors	2
[G-06]	Don't initialize variables with default value	2
[G-07]	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)	4
[G-08]	Use shift Right/Left instead of division/multiplication if possible	5

	Issue	Instances
[G-09]	Use != 0 instead of > 0 for unsigned integer comparison	5
[G-10]	Use unchecked when it is safe	?
[G-11]	Refund contribution and pending balance in the same call	1
[G-12]	Generate merkle tree offchain	1
[G-13]	Pack Bid Struck	1
[G-14]	Pack Queue Struck	1

[G-01] Use assembly to check for `address(0)`

Saves 6 gas per instance

Instances (2):

```
File: src/seaport/modules/OptimisticListingSeaport.sol

106:                proposedListings[_vault].proposer == address(0)

107:                activeListings[_vault].proposer == address(0)
```

Link to code

[G-02] Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Instances (2):

```
File: src/lib/MinPriorityQueue.sol

98:         for (uint256 i = 0; i < curUserBids.length; i++) {
```

Link to code

```
File: src/seaport/modules/OptimisticListingSeaport.sol

390:             for (uint256 i = 0; i < _offer.length; ++i) {
```

Link to code

[G-03] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (100 gas) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

Saves 100 gas per instance

Instances (5):

```
File: src/modules/GroupBuy.sol

92:         contribute(currentId, _quantity, _raePrice);
```

Link to code

```
File: src/punks/protoforms/PunksMarketBuyer.sol

37:         proxy = IWrappedPunk(wrapper).proxyInfo(address(this

64:         IERC721(wrapper).safeTransferFrom(address(this), va
```

[Link to code]

```
File: src/seaport/modules/OptimisticListingSeaport.sol

362:                seaportLister,
```

[Link to code]

```
File: src/seaport/targets/SeaportLister.sol

38:                         IERC1155(token).setApprovalForAll(co
```

[Link to code]

[G-04] Use calldata instead of memory for function arguments that do not get mutated

Mark data types as `calldata` instead of `memory` where possible. This makes it so that the data is not automatically loaded into memory. If the data passed into the function does not need to be changed (like updating values in an array), it can be passed in as `calldata`. The one exception to this is if the argument must later be passed into another function that takes an argument that specifies `memory` storage.

Instances (5):

```
File: src/modules/GroupBuy.sol

166:         bytes memory _purchaseOrder,

167:         bytes32[] memory _purchaseProof
```

[Link to code]

```
File: src/punks/protoforms/PunksMarketBuyer.sol
```

```
46:     function execute(bytes memory _order) external payable r
```

Link to code

```
File: src/seaport/targets/SeaportLister.sol

26:     function validateListing(address _consideration, Order[]

51:     function cancelListing(address _consideration, OrderComp
```

Link to code

[G-05] Use Custom Errors

Source

Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

Instances (2):

```
File: src/lib/MinPriorityQueue.sol

42:         require(!isEmpty(self), "nothing to return");

91:         require(!isEmpty(self), "nothing to delete");
```

Link to code

[G-06] Don't initialize variables with default value

Instances (2):

```
File: src/lib/MinPriorityQueue.sol

98:         for (uint256 i = 0; i < curUserBids.length; i++) {
```

[Link to code]

```
File: src/seaport/modules/OptimisticListingSeaport.sol

390:            for (uint256 i = 0; i < _offer.length; ++i) {
```

[Link to code]

[G-07] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves 5 gas per loop

Instances (4):

```
File: src/lib/MinPriorityQueue.sol

60:                 j++;

77:         insert(self, Bid(self.nextBidId++, owner, price, qua

98:         for (uint256 i = 0; i < curUserBids.length; i++) {
```

[Link to code]

```
File: src/modules/GroupBuy.sol

74:         poolInfo[++currentId] = PoolInfo(
```

[Link to code]

[G-08] Use shift Right/Left instead of division/multiplication if possible

Instances (5):

```
File: src/lib/MinPriorityQueue.sol

49:         while (k > 1 && isGreater(self, k / 2, k)) {

50:             exchange(self, k, k / 2);

51:             k = k / 2;
```

Link to code

```
File: src/seaport/modules/OptimisticListingSeaport.sol

395:         uint256 openseaFees = _listingPrice / 40;

396:         uint256 tesseraFees = _listingPrice / 20;
```

Link to code

[G-09] Use != 0 instead of > 0 for unsigned integer comparison

Instances (5):

```
File: src/modules/GroupBuy.sol

124:         if (fillAtAnyPriceQuantity > 0) {

139:         if (filledQuantity > 0) minReservePrices[_poolId] =

268:         if (pendingBalances[msg.sender] > 0) withdrawBalanc

297:         while (quantity > 0) {
```

Link to code

```
File: src/seaport/modules/OptimisticListingSeaport.sol
```

```
469:         if (isValidated && !isCancelled && totalFilled > 0
```

Link to code

[G-10] Use unchecked when it is safe

For example, those dealing with eth value will basically never overflow:

```
        userContributions[_poolId][msg.sender] += msg.value;
        totalContributions[_poolId] += msg.value;
```

Link to code

```
            filledQuantities[_poolId] += fillAtAnyPriceQuantity;
```

Link to code

Also those with explicit check before

```
                lowestBid.quantity -= quantity;
```

Link to code

[G-11] Refund contribution and pending balance in the same call

```
        // Transfers remaining contribution balance back to call
        payable(msg.sender).call{value: contribution}("");

        // Withdraws pending balance of caller if available
        if (pendingBalances[msg.sender] > 0) withdrawBalance();
```

Link to code

[G-12] Generate merkle tree offchain

Generate merkle tree onchain is expensive espically when you want to include a large set of value. Consider generating it offchain a publish the root when creating a new pool.

```
            bytes32 merkleRoot = (length == 1) ? bytes32(_tokenIds[(
```

[Link to code]

[G-13] Pack Bid Struct

The Bid struct can be packed tighter

```
    struct Bid {
        uint256 bidId;
        address owner;
        uint256 price;
        uint256 quantity;
    }
```

[Link to code]

to

```
    struct Bid {
        uint96 bidId;
        address owner;
        uint256 price;
        uint256 quantity;
    }
```

[G-14] Pack Queue Struct

```
        struct Queue {
            ///@notice incrementing bid id
            uint256 nextBidId;
            ///@notice array backing priority queue
            uint256[] bidIdList;
            ///@notice total number of bids in queue
            uint256 numBids;
            //@notice map bid ids to bids
            mapping(uint256 => Bid) bidIdToBidMap;
            ///@notice map addreses to bids they own
            mapping(address => uint256[]) ownerToBidIds;
        }
```

Link to code

to

```
    struct Queue {
        ///@notice incrementing bid id
        uint96 nextBidId;
        ///@notice total number of bids in queue
        uint160 numBids;
        ///@notice array backing priority queue
        uint256[] bidIdList;
        //@notice map bid ids to bids
        mapping(uint256 => Bid) bidIdToBidMap;
        ///@notice map addreses to bids they own
        mapping(address => uint256[]) ownerToBidIds;
    }
```

HickupHH3 (judge) commented:

> I like that #28 states the amount of gas saved per finding and as a whole. However, this report edges out in gas savings IMO, notably with the struct reorderings + doing merkle proof generation off-chain recommendations.

> *(Note: See* original submission *for judge's full commentary.)*

mehtaculous (Tessera) confirmed

Mitigation Review

Introduction

Following the C4 audit contest, 3 wardens (IllIllI, Lambda, and gzeon) reviewed the mitigations for all identified issues. Additional details can be found within the C4 Tessera Versus Mitigation Review contest repository.

Mitigation Review Scope

URL	Mitigation of	Purpose
PR 204	H-01	Fixed by checking success value of call.
PR 207	H-02	Calculate actual minReservePrice after purchase has been made. This is done by simply dividing total filled quantity by the purchase price. This fixes the ETH accounting when the NFT is purchased for less than the minimum reserve price.
PR 203	H-03	Updated comparison to use <= to avoid edge case for an invalid state.
PR 202	H-04, H-07	Updated pendingBalances mapping before setting the newly proposed listing. This way, the balance is updated for the previous proposer and the currently proposed listing can be overridden with the new one.
Not fixing	H-05	N/A
PR 211	H-06	Solution is to move orderHash to Listing struct so that active and proposed listings can have separate order hashes.
PR 201	H-08, H-09	Fixed by moving success state update before call to market buyer.
Not fixing	M-01	N/A
Not fixing	M-02	N/A
PR 206	M-03	If total supply is equal to filled quantities, use min reserved price instead of min price when validating the user contribution. this way, funds will not be stuck

URL	Mitigation of	Purpose
		until a purchase is made and the user will be able to withdraw their funds right away.
PR 212	M-04, M-08	Used <= when comparing the new price to the lowest bid price. That way, it can only override the lower bid if price is higher.
Not fixing	M-05	N/A
Not fixing	M-06	N/A
Not fixing	M-07	N/A
PR 213	M-09	Where due to truncation, the stated minimum raised wouldn't result in actually being able to purchase with the full amount raised. Instead of doing that math we're going to have the initial purchase by the deployer set the initial target.

Mitigation Review Summary

Of the mitigations reviewed, 10 have been confirmed:

- H-01: Mitigation confirmed by gzeon, IIIIIII, and Lambda.
- H-03: Mitigation confirmed by gzeon, IIIIIII, and Lambda.
- H-04: Mitigation confirmed by gzeon, IIIIIII, and Lambda.
- H-06: Mitigation confirmed by gzeon, IIIIIII, and Lambda.
- H-07: Mitigation confirmed by gzeon, IIIIIII, and Lambda.
- H-09: Mitigation confirmed by gzeon, IIIIIII, and Lambda.
- M-03: Mitigation confirmed by gzeon and Lambda.
- M-04: Mitigation confirmed by gzeon and Lambda.
- M-08: Mitigation confirmed by gzeon and Lambda.
- M-09: Mitigation confirmed by gzeon, IIIIIII, and Lambda.

The two remaining mitigations have either not been confirmed and/or introduced new issues. See full details below.

(Note: mitigation reviews below are referenced as `MR:S-N`*,* `MitigationReview:NewIssueSeverity-NewIssueNumber`*)*

[MR:H-01] Mitigation doesn't take rounding issue into account

Submitted by IllIllI, also found by gzeon.

https://github.com/fractional-company/modular-fractional/blob/fc15c85069d8d55cfe840d4c313754c77f18979f/src/modules/GroupBuy.sol#L198-L199

Vulnerability details

The PR applies the recommended mitigation from the original finding (H-02), but doesn't take into account the rounding issue identified in M-09

Impact

If the price the NFT is bought for is not an exact multiple of the `filledQuantities`, there will be a loss of precision, and during the calculation of refunds, each user will get back more than they should, which will leave the last user with not enough funds to withdraw. It is much more likely for the price *not* to be an exact multiple than it is for it to be one, so most `GroupBuy`s will have this issue.

Proof of Concept

The division will have a loss of precision:

```
            // Calculates actual min reserve price based on purchase
            minReservePrices[_poolId] = _price / filledQuantity;
```

https://github.com/fractional-company/modular-fractional/blob/fc15c85069d8d55cfe840d4c313754c77f18979f/src/modules/GroupBuy.sol#L198-L199

And the same example the judge used in finding M-9 would still apply.

Recommended Mitigation Steps

Decrease the total number of Raes if the price is not a multiple of the current total, so that it becomes a multiple.

stevennevins (Tessera) acknowledged

[MR:H-02] Attacker can steal the NFT bought by sending it to another vault he controls

Submitted by gzeon*.*

https://github.com/fractional-company/modular-fractional/blob/d0974eef922c3087ed9c1b8d758fe66307734c23/src/modules/GroupBuy.sol#L214

Impact

The mitigation of H-08 tries to validate the vault returned by _market with the VaultRegistry. However, it's only validated if the vault exists, but not if it is the correct vault. A similar attack described in https://github.com/code-423n4/2022-12-tessera-findings/issues/47 can be carried out by using a valid vault address that is permissionlessly deployed with VaultRegistry.createFor but have the owner set to the attacker.

Proof of Concept

https://github.com/fractional-company/modular-fractional/blob/d0974eef922c3087ed9c1b8d758fe66307734c23/src/modules/GroupBuy.sol#L214-L215

```
        (, uint256 id) = IVaultRegistry(registry).vaultToToken(v
        if (id == 0) revert NotVault();
```

1. Group assembles and raises funds to buy NFT X

2. Attacker deploy a valid vault using VaultRegistry.createFor but setting himself as the owner, with a merkle root that enable NFT transfer
3. Attacker calls purchase() and supplies their malicious contract in _market, returning the vault generated above
4. Attacker call Vault.execute in the vault to retrieve the NFT to his own wallet
5. Attacker receives the NFT bought by the raised funds with 0 cost except gas

https://github.com/fractional-company/modular-fractional/blob/9411afcd03d457192adc2bc59b3b378aeddd5865/src/Vault.sol#L36

```
    function execute(
        address _target,
        bytes calldata _data,
        bytes32[] calldata _proof
    ) external payable returns (bool success, bytes memory respc
        bytes4 selector;
        assembly {
            selector := calldataload(_data.offset)
        }

        // Generate leaf node by hashing module, target and func
        bytes32 leaf = keccak256(abi.encode(msg.sender, _target,
        // Check that the caller is either a module with permiss
        if (!MerkleProof.verify(_proof, MERKLE_ROOT(), leaf)) {
            if (msg.sender != FACTORY() && msg.sender != OWNER()
                revert NotAuthorized(msg.sender, _target, select
        }

        (success, response) = _execute(_target, _data);
    }
```

Recommended Mitigation Steps

Also check the owner of the vault.

HickupHH3 (judge) commented:

> It's clear to me that the vault should be the owner of the NFT. The question I have is, who should be the owner of the vault? Multisig? What address should be

> considered to be the rightful owner?

> Once the purchase is executed, a new vault gets deployed with the proper permissions, the NFT then gets transferred to the vault, and ownership of the NFT by the vault is verified.

> Based on this, given that the market is still a user supplied param, a valid vault can be deployed, but it belongs to an arbitrary owner.

stevennevins (Tessera) acknowledged

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