

Sonic Airdrop Contracts
Security Review

Cantina Managed review by:

Saw-mon and Natalie, Lead Security Researcher
Cccz, Security Researcher

July 31, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Critical Risk . . . 4
  - 3.1.1 `isBudgetConstrainedBuyingLimitReached` should be set as soon as the remaining budget cannot afford the full order cost . . . 4
- 3.2 High Risk . . . 11
  - 3.2.1 Invariants of Price Segments (*version 2*) . . . 11
- 3.3 Medium Risk . . . 25
  - 3.3.1 Unsafe node check in `_cancelOrdersSide` . . . 25
  - 3.3.2 Add invariant checks to the `_consumePriceSegments` and make it less prone to malfunction . . . 25
  - 3.3.3 Quote tokens received by the orderbook versus to-be-claimed quote tokens by sellers 28
- 3.4 Informational . . . 29
  - 3.4.1 Add invariant checks when adding and canceling orders . . . 29
  - 3.4.2 Use `quantityFulfilled` for sell market orders when transferring NFTs . . . 30

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Paintswap is an open NFT marketplace, NFT launchpad, VRF and financial NFT order book on Sonic. They are also the developers of the Web3 game Estfor Kingdom.

From Jul 25th to Jul 30th the Cantina team conducted a review of sonic-airdrop-contracts on commit hash 09a09846. The particular scope of the review consisted of the following contracts:

- `contracts/SonicOrderBook.sol`
- `contracts/libraries/SegmentLibrary.sol`
- `contracts/libraries/SlotLibrary.sol`
- `contracts/libraries/SonicOrderBookLibrary.sol`

The team identified a total of **7** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	1	1	0
Medium Risk	3	3	0
Low Risk	0	0	0
Gas Optimizations	0	0	0
Informational	2	2	0
Total	**7**	**7**	**0**

The Cantina Managed team reviewed Paintswap's `sonic-airdrop-contracts` holistically on commit hash 4264fdb3 and concluded that all findings were addressed and no new vulnerabilities were identified.

3 Findings

3.1 Critical Risk

3.1.1 `isBudgetConstrainedBuyingLimitReached` should be set as soon as the remaining budget cannot afford the full order cost

Severity: Critical Risk

Context: SonicOrderBook.sol#L845

Description: `isBudgetConstrainedBuyingLimitReached` should be set as soon as the remaining budget cannot afford the full order cost. Here is why this is an issue currently. If we can only partially afford to fill the current order to be consumed, ie `maxAffordableQuantity` (quantised) is not-zero and we partially fill the order, then the implementation moves on to the next order. Most likely in the next iteration, one cannot fill any of the next order, since the `remainingBudget` would be really small and `bestPrice` potentially could be even higher (in budget-constrained market buy orders, best prices increment up as we fill/match orders):

$$\left\lfloor \frac{\Delta q p_{b'}}{10^d} \right\rfloor > \frac{(\epsilon_0 + \epsilon_1 \Delta q) p_b}{10^d} + \epsilon_2$$

In this scenario, after the first budget-constrained partial fill, if one queries for the next order to potentially match, if it ends up matching `0` amount. We would hit this branch:

```
if (maxAffordableQuantity == 0) {
  // Can't afford any of this order, stop here
  params.isBudgetConstrainedBuyingLimitReached = true;
  return (
    segment,
    0, // ordersConsumed
    0 // quantityNFTClaimable
  );
}
```

Note that we have queried `2` orders so far and for both one `ordersConsumed` has been `0`. At this point we have:

parameter	value
`segment`	the segment that includes the last skipped order
`lastConsumedSegmentIndex`	might be potentially the same as the first order or pointing to the next segment since we have quried 2 orders
`numOrdersFullyConsumed`	does not count for the last orders: the one that was partially matched and also the one that was skipped since `maxAffordableQuantity` ends up being `0`

Now, let's analyse this portion of the `_updateSegmentsAndTree`:

```
// if the order skipped after the partial order match was on the next segement
// the value for `lastSegmentDeleted` would be `false`
bool lastSegmentDeleted = lastSegmentIndex < (oldTombstoneOffset + numSegmentsFullyConsumed);
if (!lastSegmentDeleted) {
    // `numOrdersWithinLastSegmentFullyConsumed` would be number of orders fully consumed
    // in before querying the partial or skipped order. Might be the come from the
    // segment the partial order was on or before
    if (numOrdersWithinLastSegmentFullyConsumed != 0) {
        for (uint256 i; i < numOrdersWithinLastSegmentFullyConsumed; ++i) {
            // potentailly wrong segment is updated, since the assumed fully
            // matched orders might not be on the same segment as the
            // last order which was skipped
            lastSegment = lastSegment.clearSlot(i);
        }
    }

    if (uint256(lastSegment) == 0) {
        // All orders in the last segment are consumed, delete from tree
        tree.remove(bestPrice);
        // Don't pop segment for gas efficiency, instead just write zeros
    }

    // segments gets updated incorrectly
    segments[lastSegmentIndex] = lastSegment;
}
```

Let's follow all this with an example from transaction 0xd1945911cce9d105b4503d3ab378e735c754ae1c5d5df62fe4b824b884c2b3c1:

```
(10, 40416) (10, 40415) (10, 40414)
(10, 40419) (10, 40418) (10, 40417)
```

with `maxCost` $c_{max} = 31500$. The transaction fully consumed the orders `40414` and `40415` and at this point the `params.currentCost` would be $c_{curr} = 31490$ and the remaining budget would be $c_r = 10$, the quantised `maxAffordableQuantity` $q_{max} = 2\Delta q$ (Δq being the `QUANTITY_TICK`). Order `40416` gets partialy filled and the slot (in stack) gets updated to `(8, 40416)`. Since we have not set the `isBudgetConstrainedBuyingLimitReached` and the other conditions are satisfied (there is a remaining budget and we have not hit the limit price, ...), querying over the orders continues. At this point we have $c_{curr} = 31498$ and thus $c_r = 2$ this time q_{max} ends up being 0 and this `isBudgetConstrainedBuyingLimitReached` gets set. The value of `segment` ends up being:

```
(10, 40419) (10, 40418) (10, 40417) segment
```

The storage update in `_updateSegmentsAndTree` performs cleaning on the above segment and since `numOrdersFullyConsumed` did not account for the partial order and the skipped order the `tombstone` also lags behind and we end up with the following segment structure:

```
(10, 40416) (10, 40415) (10, 40414) <-- tombstone offset
(10, 40419) ( 0,     0) ( 0,     0)
```

Note that instead of clearning out the orders `40415` and `40414` which were fully matched the orders `40418` and `40417` get cleared out and the order `40416` does not even get updated although it was partially matched.

Proof of Concept: Apply the following patch which will be needed in the test cases:

```
  diff --git a/test/sonicOrderBook/utils.ts b/test/sonicOrderBook/utils.ts
  index be0608f..845dea0 100644
-  -- a/test/sonicOrderBook/utils.ts
+  ++ b/test/sonicOrderBook/utils.ts
  @@ -59,6 +59,7 @@ export async function deployContractsFixture() {

     return {
       orderBook,
+      sonicOrderBookLibrary,
       sonicAirdropNFT,
       wS,
       quoteToken,
```

Then create the file `test/sonicOrderBook/isBudgetConstrainedBuyingLimitReached.bug.ts` with the following content:

```
import {loadFixture} from "@nomicfoundation/hardhat-toolbox/network-helpers";
import hre, {ethers, upgrades} from "hardhat";
import {expect} from "chai";
import {SonicOrderBook} from "../../typechain-types";
import {Order, OrderSide} from "@paintswap/sonic-airdrop-definitions/types";
import {deployContractsFixture, calcFees, ONE_ETHER, RETURN_WRAPPED_SONIC} from "./utils";
import {initializerSlot} from "../utils";
import {BigNumberish, Block} from "ethers";
import {deploySonicOrderBook, deploySonicOrderBookLibrary} from "../../scripts/deployHelpers";
import { HardhatEthersSigner } from "@nomicfoundation/hardhat-ethers/signers";

// Uses a plain ERC20 token with 18 decimals
describe("SonicOrderBook", function () {
  describe("Market orders with useExactQuantity: false", function () {
    describe("Budget-constrained buying", function () {
      async function setup() {
        const {
          sonicAirdropNFT,
          wS,
          sonicOrderBookLibrary,
          quoteToken,
          alice,
          bob,
          dev,
          tokenId,
          initialCoins,
          maxOrdersPerPrice,
          QUANTITY_TICK
        } = await loadFixture(deployContractsFixture);

        await quoteToken.connect(bob).mint(bob, initialCoins);

        const orderBook = await deploySonicOrderBook(sonicAirdropNFT, quoteToken, wS, sonicOrderBookLibrary);

        // set approvals
        await quoteToken.approve(orderBook, initialCoins);
        await quoteToken.connect(alice).approve(orderBook, initialCoins);
        await sonicAirdropNFT.setApprovalForAll(orderBook, true);
        await sonicAirdropNFT.connect(alice).setApprovalForAll(orderBook, true);
        await quoteToken.connect(bob).approve(orderBook, initialCoins);

        const priceTick = 100n;
        const minQuantity = 10n * QUANTITY_TICK;
        const isTradeable = true;

        const makerFeeBps = 40n; // 0.2% fee for makers
        const takerFeeBps = 60n; // 0.3% fee for takers
        await orderBook.setAdminVars(
            dev,
            makerFeeBps,
            takerFeeBps,
            maxOrdersPerPrice,
            [tokenId],
            [{priceTick, minQuantity, isTradeable}]
        );

        return {
          orderBook,
          quoteToken,
          alice,
          bob,
          dev,
          tokenId,
          initialCoins,
          maxOrdersPerPrice,
          QUANTITY_TICK,
          priceTick,
          minQuantity
        }
      }

      it("6 orders", async function () {
        const {
```

```
    orderBook,
    quoteToken,
    alice,
    bob,
    tokenId,
    maxOrdersPerPrice,
    QUANTITY_TICK,
    priceTick,
    minQuantity
  } = await loadFixture(setup);

  const limitPrice = 4n * priceTick;
  const quantity = minQuantity;
  const totalOrders = 6;

  await placeNLimitOrders(
    orderBook,
    alice,
    totalOrders,
    OrderSide.Sell,
    tokenId,
    limitPrice,
    quantity,
    true,
    true
  );

  await logSegments(orderBook, OrderSide.Sell, tokenId, limitPrice, "segments before calling
  ↪  placeMarketOrder :");

  const nodeBefore = await orderBook.getNode(OrderSide.Sell, tokenId, limitPrice);
  expect(nodeBefore.tombstoneSegmentOffset).to.be.equal(0);

  const segmentsBefore = await getSegments(orderBook, OrderSide.Sell, tokenId, limitPrice);
  expect(segmentsBefore[0]).to.be.equal("0x00000000000a0000000000030000000a0000000000020000000a000000000
  ↪  001");
  expect(segmentsBefore[1]).to.be.equal("0x00000000000a0000000000060000000a0000000000050000000a000000000
  ↪  004");

  // each order is worth 40, so in total with a budget of 101
  // we should afford 2 full orders and a partial order (1/2 of an order)
  // this would cost 100, the extra budget of 1 is here to make sure
  // after we partially match the 3rd order we would not reach the `maxCost`
  // so the loop would continue to trigger the bug
  const budget = 101;
  await orderBook.connect(bob).placeMarketOrder(
    {
      side: OrderSide.Buy,
      tokenId,
      totalCost: budget,
      quantity: 0,
      useExactQuantity: false
    },
    RETURN_WRAPPED_SONIC
  );

  await logSegments(orderBook, OrderSide.Sell, tokenId, limitPrice, "segments after calling
  ↪  placeMarketOrder :");

  const nodeAfter = await orderBook.getNode(OrderSide.Sell, tokenId, limitPrice);
  expect(nodeAfter.tombstoneSegmentOffset).to.be.equal(0);

  const segmentsAfter= await getSegments(orderBook, OrderSide.Sell, tokenId, limitPrice);
  expect(segmentsAfter[0]).to.be.equal("0x0000000000050000000000030000000000000000000000000000000000000
  ↪  00");
  expect(segmentsAfter[1]).to.be.equal("0x00000000000a0000000000060000000a0000000000050000000a000000000
  ↪  04");

});

it("4 orders", async function () {
  const {
    orderBook,
    quoteToken,
    alice,
    bob,
```

```
        tokenId,
        maxOrdersPerPrice,
        QUANTITY_TICK,
        priceTick,
        minQuantity
      } = await loadFixture(setup);

      const limitPrice = 4n * priceTick;
      const quantity = minQuantity;
      const totalOrders = 4;

      await placeNLimitOrders(
        orderBook,
        alice,
        totalOrders,
        OrderSide.Sell,
        tokenId,
        limitPrice,
        quantity,
        true,
        true
      );

      await logSegments(orderBook, OrderSide.Sell, tokenId, limitPrice, "segments before calling
      ↪  placeMarketOrder :");

      const nodeBefore = await orderBook.getNode(OrderSide.Sell, tokenId, limitPrice);
      expect(nodeBefore.tombstoneSegmentOffset).to.be.equal(0);

      const segmentsBefore = await getSegments(orderBook, OrderSide.Sell, tokenId, limitPrice);
      expect(segmentsBefore[0]).to.be.equal("0x00000000000a000000000003000000a000000000020000000a000000000
      ↪  001");
      expect(segmentsBefore[1]).to.be.equal("0x0000000000000000000000000000000000000000000000000a000000000
      ↪  004");

      // each order is worth 40, so in total with a budget of 101
      // we should afford 2 full orders and a partial order (1/2 of an order)
      // this would cost 100, the extra budget of 1 is here to make sure
      // after we partially match the 3rd order we would not reach the `maxCost`
      // so the loop would continue to trigger the bug
      const budget = 101;
      await orderBook.connect(bob).placeMarketOrder(
        {
          side: OrderSide.Buy,
          tokenId,
          totalCost: budget,
          quantity: 0,
          useExactQuantity: false
        },
        RETURN_WRAPPED_SONIC
      );

      await logSegments(orderBook, OrderSide.Sell, tokenId, limitPrice, "segments after calling
      ↪  placeMarketOrder :");

      const nodeAfter = await orderBook.getNode(OrderSide.Sell, tokenId, limitPrice);
      expect(nodeAfter.tombstoneSegmentOffset).to.be.equal(0);

      const segmentsAfter= await getSegments(orderBook, OrderSide.Sell, tokenId, limitPrice);
      expect(segmentsAfter[0]).to.be.equal("0x00000000000500000000000300000000000000000000000000000000000
      ↪  00");
      expect(segmentsAfter[1]).to.be.equal("0x0000000000000000000000000000000000000000000000000a000000000
      ↪  04");
    });
  });
});

async function getNextOrderId(orderBook: SonicOrderBook) {
  const nextOrderIdSlot = 0;
  const packedSlot = await ethers.provider.getStorage(orderBook, nextOrderIdSlot);
  return parseInt(packedSlot.slice(2, 14), 16);
}

async function placeNLimitOrders(
  orderbook: SonicOrderBook,
  user: HardhatEthersSigner,
```

```
    totalOrders: number,
    side: OrderSide,
    tokenId: BigNumberish,
    price: BigNumberish,
    quantity: BigNumberish,
    onlyPost: boolean,
    onlyExactPriceIfMaker: boolean
  ) {
    for(let i = 0; i < totalOrders; i++) {
      await orderbook.connect(user).placeLimitOrders([
        {
          side: side,
          tokenId: tokenId,
          price: price,
          quantity,
          onlyPost: onlyPost,
          onlyExactPriceIfMaker: onlyExactPriceIfMaker
        }
      ],
        RETURN_WRAPPED_SONIC
      );
    }
  }

  async function logSegments(orderBook: SonicOrderBook, side: OrderSide, tokenId: BigNumberish, price:
  ↪  BigNumberish, message: string) {
    const segmentsAfter = await getSegments(orderBook, side, tokenId, price);
    console.log(message)
    console.log(segmentsAfter);
  }

  async function getSegments(orderBook: SonicOrderBook, side: OrderSide, tokenId: BigNumberish, price:
  ↪  BigNumberish) {
    // Storage slot assignments based on contract layout:
    // slot 0: packed variables (_devAddr, _devFeeBps, _maxOrdersPerPrice, _nextOrderId)
    // slot 1: _tokenIdInfos mapping
    // slot 2: _asks mapping
    // slot 3: _bids mapping
    // slot 4: _asksAtPrice mapping
    // slot 5: _bidsAtPrice mapping

    const baseSlot = side === 0 ? 5 : 4; // 0 = Buy (bids), 1 = Sell (asks)

    // Calculate slot for tokenId in the outer mapping
    const tokenIdSlot = ethers.solidityPackedKeccak256(["uint256", "uint256"], [tokenId, baseSlot]);

    // Calculate slot for price in the inner mapping
    const priceSlot = ethers.solidityPackedKeccak256(["uint256", "bytes32"], [price, tokenIdSlot]);

    // Read array length from the calculated slot
    const lengthHex = await ethers.provider.getStorage(orderBook, priceSlot);
    const length = parseInt(lengthHex, 16);

    if (length === 0) {
      return [];
    }

    // For dynamic arrays, elements start at keccak256(slot)
    const arrayStartSlot = ethers.solidityPackedKeccak256(["bytes32"], [priceSlot]);

    // Read each segment (bytes32 elements)
    const segments = [];
    for (let i = 0; i < length; i++) {
      const elementSlot = BigInt(arrayStartSlot) + BigInt(i);
      const segment = await ethers.provider.getStorage(orderBook, elementSlot);
      segments.push(segment);
    }

    return segments;
  }
});
```

Then run:

```
yarn test -- test/sonicOrderBook/isBudgetConstrainedBuyingLimitReached.bug.ts
```

The tests should fail without the fix applied and pass with the fixes applied. Logs:

```
segments before calling placeMarketOrder :
[
  '0x00000000000a0000000000030000000a0000000000020000000a000000000001',
  '0x00000000000a0000000000060000000a0000000000050000000a000000000004'
]
segments after calling placeMarketOrder :
[
  '0x00000000000a0000000000030000000a0000000000020000000a000000000001',
  '0x00000000000a0000000000060000000000000000000000000000000000000000'
]
        1) 6 orders
segments before calling placeMarketOrder :
[
  '0x00000000000a0000000000030000000a0000000000020000000a000000000001',
  '0x0000000000000000000000000000000000000000000000000000000a000000000004'
]
segments after calling placeMarketOrder :
[
  '0x00000000000a0000000000030000000a0000000000020000000a000000000001',
  '0x0000000000000000000000000000000000000000000000000000000000000000'
]
        2) 4 orders


  0 passing (321ms)
  2 failing

  1) SonicOrderBook
       Market orders with useExactQuantity: false
         Budget-constrained buying
           6 orders:

      AssertionError: expected '0x00000000000a0000000000030000000a000' to equal
      ↪  '0x000000000005000000000030000000000'
      + expected - actual

      -0x00000000000a0000000000030000000a0000000000020000000a000000000001
      +0x000000000005000000000030000000000000000000000000000000000000000

      at Context.<anonymous> (test/sonicOrderBook/isBudgetConstrainedBuyingLimitReached.bug.ts:133:40)

  2) SonicOrderBook
       Market orders with useExactQuantity: false
         Budget-constrained buying
           4 orders:
     Error: VM Exception while processing transaction: reverted with custom error 'KeyDoesntExist()'
    at SonicOrderBook.getNode (contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol:71)
    at SonicOrderBook.getNode (contracts/SonicOrderBook.sol:519)
    at <UnrecognizedContract>.<unknown> (0xa51c1fc2f0d1a1b8494ed1fe312d7c3a78ed91c0)
    at EdrProviderWrapper.request
    ↪  (node_modules/hardhat/src/internal/hardhat-network/provider/provider.ts:398:41)
    at async staticCallResult (node_modules/ethers/src.ts/contract/contract.ts:337:22)
    at async staticCall (node_modules/ethers/src.ts/contract/contract.ts:303:24)
    at async Proxy.getNode (node_modules/ethers/src.ts/contract/contract.ts:351:41)
    at async Context.<anonymous> (test/sonicOrderBook/isBudgetConstrainedBuyingLimitReached.bug.ts:195:27)
```

Recommendation: Apply the following patch to stop querying more orders as soon as one cannot afford a full order in the budget-constrained market buy order:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index 31b5486..1a4c724 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -836,13 +836,13 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
          uint256 remainingBudget = params.maxCost - params.currentCost;

          if (fullOrderCost > remainingBudget) {
+           // Can't afford a full order, stop here
+           params.isBudgetConstrainedBuyingLimitReached = true;
            // Check if we can afford partial order
            uint256 maxAffordableQuantity = (remainingBudget * BASE_DECIMAL_DIVISOR) / params.bestPrice;
            // Floor to QUANTITY_TICK to ensure proper alignment
            maxAffordableQuantity = (maxAffordableQuantity / QUANTITY_TICK) * QUANTITY_TICK;
            if (maxAffordableQuantity == 0) {
-             // Can't afford any of this order, stop here
-             params.isBudgetConstrainedBuyingLimitReached = true;
              return (
                segment,
                0, // ordersConsumed
```

Paintswap: Fixed in PR 106.

Cantina Managed: Fix verified.

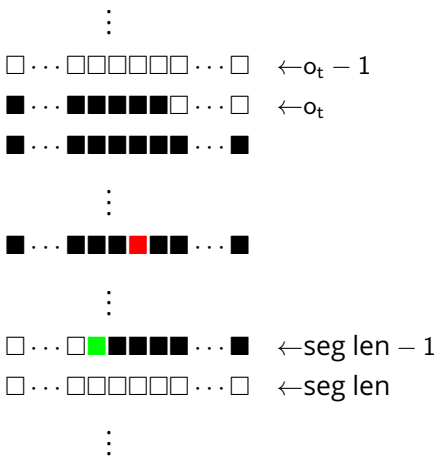
3.2 High Risk

3.2.1 Invariants of Price Segments (*version 2*)

Severity: High Risk

Context: SonicOrderBook.sol#L845, SonicOrderBook.sol#L909-L952

Description:



- □ represents empty orders with 0 order id and normalized quantity.
- ■ represent orders where both the order id and the normalized quantity are non-zero.

Invariants:

1. The following configuration in the active area of the segments when all orders are drawn on a line should also not be allowed $\cdots$■□■$\cdots$, it should be prxoved that this cannot happen. We call these empty slots holes.
2. If an order has an order id 0 then its normalized quantity should also be 0 (is not allowed) and vice versa (is not allowed).
3. Orders are inserted/added at the end.
4. Orders are matched based on their order id. The smaller order ids get matched first (FIFO).
5. The last segment of an existing price node $p \in T_{tid,s}$ has at least one non-empty order:

$$\cdots\blacksquare\cdots \leftarrow \text{seg len} - 1$$

For the case of 3 slots per segment the following cases are possible:

$$\blacksquare\blacksquare\blacksquare \leftarrow \text{seg len} - 1$$
$$\square\blacksquare\blacksquare \leftarrow \text{seg len} - 1$$
$$\square\square\blacksquare \leftarrow \text{seg len} - 1$$
$$\blacksquare\blacksquare\square \leftarrow \text{seg len} - 1 = o_t$$
$$\square\blacksquare\square \leftarrow \text{seg len} - 1 = o_t$$
$$\blacksquare\square\square \leftarrow \text{seg len} - 1 = o_t$$

6. The segment pointed to by o_t (`tombstoneOffset`) if the price node exists $p \in T_{tid,s}$ in the tree, then this segment would have at least one non-empty order:

$$\cdots\blacksquare\cdots \leftarrow o_t$$

For the case of 3 slots per segment the following cases are possible:

$$\blacksquare\blacksquare\blacksquare \leftarrow o_t$$
$$\blacksquare\blacksquare\square \leftarrow o_t$$
$$\blacksquare\square\square \leftarrow o_t$$
$$\square\blacksquare\blacksquare \leftarrow o_t = \text{seg len} - 1$$
$$\square\blacksquare\square \leftarrow o_t = \text{seg len} - 1$$
$$\square\square\blacksquare \leftarrow o_t = \text{seg len} - 1$$

7. For price nodes that do not belong to the tree $p \notin T_{tid,s}$, we should have the following configuration:

$$\square\cdots\square \leftarrow o_t = \text{seg len}$$

i.e. all the orders belonging to the segment pointed to by o_t, they should all be empty orders.

8. Either $o_t = 0$ or if $o_t > 0$ all the slots before o_t are empty slots:

$$\vdots$$
$$\square\square\square$$
$$\square\square\square \leftarrow o_t - 1$$

9. Orders belonging to segments pointed to by $k \geq$ seg len should all be empty orders (unless there is a storage collision if k is a really big number). This is a more general invariant compared to Inv 8.:

$$\square\cdots\square \leftarrow k \geq \text{seg len}$$

10. The active orders in the price segments are arranged in a monotonically increasing fashion based on their order ids (this is a more stricted version of Inv 1.).

method	Inv 1.	Inv 2.	Inv 3.	Inv 4.	Inv 5.	Inv 6.	Inv 7.	Inv. 8	Inv. 9	Inv 10.
canceling orders	✓	✓	N/A	N/A	✓	✓	✓	✓	✓	✓
placing limit orders	✓	✓	✓	N/A	✓	✓	N/A	✓	✓	✓
matching orders			N/A			X				

The ✓ above means that if the invariant was satisfied before calling the method, it will be satisfied after the call has been done (in some edge cases it would only mean the invariant is satisfied after the call has been done, but it might not be before the call for example at the very beginning.).

- **Canceling orders:** We should have $p \in T_{tid,s}$ otherwise the call should fail. First the order's position in the price segment should be found using `_find` The `_find` flow relies on the fact that there are no holes (inv 1.) in the price segments and orders are placed in a monotonically increasing fashion (Inv 10.). The invariant checks need to be performed for each branch in `_cancelOrder`:

```
if (isFinalSegment && isOnlyOrderInSegment) {
    // branch A
} else {
    // Branch B: this branch has many sub-branches
    for (/* ... */) {
        if (nextSlot == 0) {
            // this branch should only be hit on the last segment,
            // ie, when `currentOrNextSegment == segments.length - 1`
            // one can add a require statement.
        }
    }

    // Branch B1: `for` loop gets skipped (`absoluteSlotIndexToRemove == totalOrders`)
    // Branch B2: `for` loop is entered   (`absoluteSlotIndexToRemove  < totalOrders`)
}
```

method	Inv 1.	Inv 2.	Inv 3.	Inv 4.	Inv 5.	Inv 6.	Inv 7.	Inv. 8	Inv. 9	Inv 10.
canceling orders (branch A)	✓	✓	N/A	N/A	✓	✓	✓	✓	✓	✓
canceling orders (branch B1)	✓	✓	N/A	N/A	✓	✓	N/A	✓	✓	✓
canceling orders (branch B2)	✓	✓	N/A	N/A	✓	✓	N/A	✓	✓	✓

- **Placing limit orders** (adding to the book): It is very important that:
  1. `_tokenIdInfos[order.tokenId].minQuantity` $q^{tid}_{min} \neq 0$. When an admin makes a token id tid tradable it has been checked that $\Delta q | q^{tid}_{min}$ and $\Delta q \leq q^{tid}_{min}$.
  2. q_o `order.quantity` is quantised by Δq and is non-zero. This has been checked.
  3. `quantityFulfilled` q_f is also quantised by Δq. This can be proved by induction/recursion in `_parseOrder`.

  And thus one can deduce that in the branch where `_addToBook(...)` is called, `quantityRemaining` q_r is also quantised by Δq and is non-zero.

  This is very important as it would guarantee one would not add slots to the price segment with a non-zero order id but with 0 quantised/normalised quantities, ie adding a slot of the form $(0, i)$ should not be possible as this would break (Inv 2.).

```
_addOrderToEndOfSegments(/*...*/) private {
    if (segments.length != 0) {
        if (lastSegmentIndex >= node.tombstoneSegmentOffset) {
            // Branch A: If there is space at the end of last segment
            //           and if this last segment is in the active aread of the price segment
            //           add the order slot here and `return`
        }
    }

    // Branch B: push a new segment [(0,0), (0,0), (qn, oid)] to price segment
    //           as the last segment is full.
    bytes32 segment = SlotLibrary.createSlotAsBytes32(orderId, normalizedQuantity);
    segments.push(segment);
}
```

method	Inv 1.	Inv 2.	Inv 3.	Inv 4.	Inv 5.	Inv 6.	Inv 7.	Inv. 8	Inv. 9	Inv 10.
placing limit orders (branch A)	✓	✓	✓	N/A	✓	✓	N/A	✓	✓	✓
placing limit orders (branch B)	✓	✓	✓	N/A	✓	✓	N/A	✓	✓	✓

- **Matching orders:** If the last segment is not deleted but all its orders get consumed, this last segment is not popped out instead just set to `bytes32(0)` □□□ (SonicOrderBook.sol#L948-L950).

The invariants broken X.

Recommendation: Invariant checks after applying the suggested patches:

method	Inv 1.	Inv 2.	Inv 3.	Inv 4.	Inv 5.	Inv 6.	Inv 7.	Inv. 8	Inv. 9	Inv 10.
canceling orders	✓	✓	N/A	N/A	✓	✓	✓	✓	✓	✓
placing limit orders	✓	✓	✓	N/A	✓	✓	N/A	✓	✓	✓
matching orders		✓	N/A	✓	✓	✓	✓	✓	✓	✓

Matching orders: The fix for "`isBudgetConstrainedBuyingLimitReached` should be set as soon as the remaining budget cannot afford the full order cost" needs to be applied as well is:

- **Inv 6.:** Currently the Inv 6. is broken, apply the patch below to fix:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index 31b5486..e690a23 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -108,6 +108,10 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
    error SellMustUseExactQuantity();
    error MakerPriceMustMatch(uint256 expectedPrice, uint256 actualPrice);

+    // Errors related to price segment invariants being broken
+    error PriceSegmentHasAHole();
+    error IncorrectDiffBetweenLastSegmentAndTheNewTombstoneOffset();
+
    enum OrderSide {
      Buy,
      Sell
  @@ -915,39 +919,126 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder,
  ↪  R
      uint256 lastSegmentIndex,
      uint256 numOrdersFullyConsumed
    ) private {
+      // `segments.length > lastSegmentIndex >= oldTombstoneOffset`
+      // `numOrdersFullyConsumed` should be the number of order slots fully consumed
+      // yet not modified but the cursor passed over it to the next order slot.
+
+      // Assumptions:
+      //    1. `numSegmentsFullyConsumed` is calculated correctly, it should be the number of
+      //       slots fully consumed or skipped if the slot was EMPTY (0)
+      //    2. `lastSegmentIndex` points to the last segment consumed
+      //    3. `lastSegment` is the segment pointed to by `lastSegmentIndex`
+
+      // [CHECK LIST] One should enforce the following for `lastSegment` (the last segment consumed):
+      // (o: an EMPTY slot, x: a non-EMPTY slot)
+      //
+      // o o o - last consumed segment - ALLOWED (only if the price node gets removed from the tree)
+      // o o x - last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to this slot too)
+      // o x o - last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to this slot too)
+      // o x x - last consumed segment - ALLOWED (check `seg.len - 1` point to this slot too)
+      // x o o - last consumed segment - ALLOWED (check `ot` point to this slot too)
+      // x o x - last consumed segment - CATCH - no hole in the middle allowed
+      // x x o - last consumed segment - ALLOWED (check `ot` point to this slot too)
+      // x x x - last consumed segment - ALLOWED
+
      uint256 numSegmentsFullyConsumed = numOrdersFullyConsumed / NUM_SLOTS_PER_SEGMENT;
-      uint256 numOrdersWithinLastSegmentFullyConsumed = numOrdersFullyConsumed % NUM_SLOTS_PER_SEGMENT;
+      uint256 oldTombstoneOffset = bestPriceNode.tombstoneSegmentOffset;
+      uint256 newTombstoneSegmentOffset = oldTombstoneOffset + numSegmentsFullyConsumed;

-      uint56 oldTombstoneOffset = bestPriceNode.tombstoneSegmentOffset;
-      if (numSegmentsFullyConsumed != 0) {
-        // Cleanup segments to get a gas refund
-        for (uint256 i = 0; i < numSegmentsFullyConsumed; ++i) {
-          delete segments[oldTombstoneOffset + i];
-        }
+      if (lastSegmentIndex == newTombstoneSegmentOffset) {
+        // At this branch we know `segments.length > lastSegmentIndex >= newTombstoneSegmentOffset`
+        uint256 numOrdersWithinLastSegmentFullyConsumed = numOrdersFullyConsumed %
↪  NUM_SLOTS_PER_SEGMENT;
+        lastSegment = lastSegment.clearUpToSlot(numOrdersWithinLastSegmentFullyConsumed);
```

```
-       uint256 newTombstoneSegmentOffset = oldTombstoneOffset + numSegmentsFullyConsumed;
-       tree.edit(bestPrice, uint56(newTombstoneSegmentOffset));
+       if (uint256(lastSegment) == 0) {
+         // `lastSegment` has this property/assumption that either:
+         // A1. An other was partially matched in this segment or
+         // A2. We had reached the last segment in the active area of the price segments which
+         //     has some zero slots (`if (remainingSegment == 0)` branch in `_consumePriceSegments`)
+         //
+         // Moreover, there is an assumption here that if `lastSegment` that was touched during the
+         // matching/filling process ends up being `0` then that would mean, there are no orders
+         // left in the price segments since A1 would not be possible and so for A2:
+         // 1. any order slot before this segment should have been fully filled/matched and thus
+         //    should have been zeroed out in the previous `if (numSegmentsFullyConsumed != 0)`
+         //    block.
+         // 2. There could not be any non-zero order slot past this segment since otherwise,
+         //    that would indicate there would have been holes in the price segments previously
+         //    breaking (Inv 1.)
+         newTombstoneSegmentOffset = lastSegmentIndex + 1;
+         // so we would have:
+         // `segments.length >= newTombstoneSegmentOffset == lastSegmentIndex + 1`
+         // `newTombstoneSegmentOffset == lastSegmentIndex + 1 > oldTombstoneOffset`
+         // thus below `lastSegment` shuold get deleted
+
+         // o o o - last consumed segment - ALLOWED - it will get deleted and the new tombstone offset
+         //                                           points to the following segment
+
+         // The checks from the [CHECK LIST] are not required since if one hits this branch and
↪  continue
+         // without reverting, the last segment consumed would be removed from the price segments
+         // and the new tombstone offset would point to the following segment.
+       } else {
+         // At this branch we know `segments.length > lastSegmentIndex >= newTombstoneSegmentOffset`
+         // and `lastSegment` is non-zero, so at the end `bestPrice` would not get removed from the
↪  `tree`

-       // Consumed all orders at this price level, so remove it from the tree
-       if (newTombstoneSegmentOffset == segments.length) {
-         tree.remove(bestPrice);
-       }
-     }

-     bool lastSegmentDeleted = lastSegmentIndex < (oldTombstoneOffset + numSegmentsFullyConsumed);
-     if (!lastSegmentDeleted) {
-       if (numOrdersWithinLastSegmentFullyConsumed != 0) {
-         for (uint256 i; i < numOrdersWithinLastSegmentFullyConsumed; ++i) {
-           lastSegment = lastSegment.clearSlot(i);
+         // s2 s1 s0 : cs(bits) : lastSegment <-- (lastSegmentIndex == newTombstoneSegmentOffset)
+         // --------------------------------(si: the `i`th slot, cs: compressed status)-------
+         //  o  o  o :      111 : last consumed segment - ALLOWED (only if the price node gets removed
↪  from the tree)
+         //  o  o  x :      110 : last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to
↪  this slot too)
+         //  o  x  o :      101 : last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to
↪  this slot too)
+         //  o  x  x :      100 : last consumed segment - ALLOWED (check `seg.len - 1` point to this
↪  slot too)
+         //  x  o  o :      011 : last consumed segment - ALLOWED (check `ot` point to this slot too)
+         //  x  o  x :      010 : last consumed segment - CATCH - no hole in the middle allowed
+         //  x  x  o :      001 : last consumed segment - ALLOWED (check `ot` point to this slot too)
+         //  x  x  x :      000 : last consumed segment - ALLOWED
+
+         uint256 compressedStatus = lastSegment.compressedStatus();
+
+         //  o  o  o : 111 : one would not enter this branch for this case
+
+         //  x  o  x : 010 : last consumed segment - CATCH - no hole in the middle allowed
+         require(compressedStatus != 2, PriceSegmentHasAHole());
+
+         if (compressedStatus > 3) {
+           //  o  x  x : 100 : last consumed segment - ALLOWED (check `seg.len - 1` point to this slot
↪  too)
+           //  o  x  o : 101 : last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to
↪  this slot too)
+           //  o  o  x : 110 : last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to
↪  this slot too)
```

```
+            require(lastSegmentIndex == segments.length - 1, PriceSegmentHasAHole());
          }
-        }

-        if (uint256(lastSegment) == 0) {
-          // All orders in the last segment are consumed, delete from tree
-          tree.remove(bestPrice);
-          // Don't pop segment for gas efficiency, instead just write zeros
+          segments[lastSegmentIndex] = lastSegment;
        }
-        segments[lastSegmentIndex] = lastSegment;
+      } else {
+        // if `lastSegmentIndex =/= newTombstoneSegmentOffset` that would mean the matching process had
↪  ended up
+        // with ( - dashes the slot can be 0 or not, c means that it was consumed):
+        // ...
+        // x(c) -(c) -(c) : lastSegmentIndex
+        // where in the last segment, the last slot was fully filled and thus we would end up with:
+        // ...
+        // x(c) -(c) -(c) : lastSegmentIndex
+        //   -  -    -    : newTombstoneSegmentOffset
+        // So these values can only be off by `1`
+        require(lastSegmentIndex + 1 == newTombstoneSegmentOffset,
↪  IncorrectDiffBetweenLastSegmentAndTheNewTombstoneOffset());
+
+        // The checks from the [CHECK LIST] are not required since if one hits this branch and continue
+        // without reverting, the last segment consumed would be removed from the price segments
+        // and the new tombstone offset would point to the following segment.
+      }
+
+      // Cleanup segments to get a gas refund
+      for (uint256 i = oldTombstoneOffset; i < newTombstoneSegmentOffset; ++i) {
+        delete segments[i];
+      }
+
+      if (newTombstoneSegmentOffset != oldTombstoneOffset) {
+        tree.edit(bestPrice, uint56(newTombstoneSegmentOffset));
+      }
+
+      // below one can add an extra branch for `newTombstoneSegmentOffset > segments.length`
+      // and possibly revert since that would mean some invariant was broken.
+      if (newTombstoneSegmentOffset >= segments.length) {
+        tree.remove(bestPrice);
      }
    }

  diff --git a/contracts/libraries/SegmentLibrary.sol b/contracts/libraries/SegmentLibrary.sol
  index d884717..e7c91a2 100644
-  -- a/contracts/libraries/SegmentLibrary.sol
+  ++ b/contracts/libraries/SegmentLibrary.sol
  @@ -43,6 +43,39 @@ library SegmentLibrary {
      return bytes10(uint80((uint256(segment) >> (slotIndex * SEGMENT_SIZE)) & SEGMENT_MASK));
    }

+    /// @notice Compresses the segment into 3 indicator bits
+    /// @param segment The segment to read from
+    /// @return status with only 3 bits sets each indicating whether the corresponding
+    //                 slot in the segment is EMPTY or not.
+    function compressedStatus(bytes32 segment) internal pure returns (uint256 status) {
+      uint256 MASK = SEGMENT_MASK;
+      uint256 SIZE = SEGMENT_SIZE;
+
+      assembly ("memory-safe") {
+        // s2 s1 s0 : segment
+        status := iszero(and(segment, MASK))
+        segment := shr(SIZE, segment)
+
+        //  0 s2 s1 : segment
+        status := or(
+          shl(1, iszero(and(segment, MASK))),
+          status
+        )
+        segment := shr(SIZE, segment)
+
+        //  0  0 s2 : segment
+        status := or(
```

```
+          shl(2, iszero(and(segment, MASK))), // still mask in case segment has dirty uppper bits
+          status
+        )
+
+        // b2 b1 b0 : status bits (it would only have 3 set bits)
+        // b2       : is 1 if s2 the 3rd slot of original segment was EMPTY
+        //    b1    : is 1 if s1 the 2nd slot of original segment was EMPTY
+        //       b0 : is 1 if s0 the 1st slot of original segment was EMPTY
+      }
+    }
+
     /// @notice Checks if the segment only contains data in the specified slot
     /// @dev Returns true if all other slots in the segment are empty (zero)
     /// @param segment The segment to check
  @@ -60,6 +93,19 @@ library SegmentLibrary {
      return bytes32(uint256(segment) & ~(uint256(SEGMENT_MASK) << (slotIndex * SEGMENT_SIZE)));
    }

+    /// @notice Clears slots up to the specified index, setting all its bits to zero
+    /// @param segment The segment to modify
+    /// @param slotIndex The index of the slot to clear up to (0, 1, or 2)
+    /// @return newSegment The updated segment with the right most slots cleared up to `slotIndex`
+    function clearUpToSlot(bytes32 segment, uint256 slotIndex) internal pure returns (bytes32
↪  newSegment) {
+      if (slotIndex == 0) {
+        return segment;
+      }
+
+      uint256 shiftAmount = slotIndex * SEGMENT_SIZE;
+      return bytes32((uint256(segment) >> (shiftAmount)) << shiftAmount);
+    }
+
     /// @notice Returns the remaining segment data after the specified slot index
     /// @dev Shifts the segment right to remove all slots up to and including the specified index.
     ///      Useful for iterating through higher-indexed slots or processing remaining data.
```

This patch makes reasoning about other invariants easier due to the modification for `_updateSegmentsAndTree`.

- Modifications of `_updateSegmentsAndTree` with comments removed:

```
function _updateSegmentsAndTree(
  BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
  BokkyPooBahsRedBlackTreeLibrary.Node storage bestPriceNode,
  bytes32[] storage segments,
  uint64 bestPrice,
  bytes32 lastSegment,
  uint256 lastSegmentIndex,
  uint256 numOrdersFullyConsumed
) private {

  uint256 numSegmentsFullyConsumed = numOrdersFullyConsumed / NUM_SLOTS_PER_SEGMENT;
  uint256 oldTombstoneOffset = bestPriceNode.tombstoneSegmentOffset;
  uint256 newTombstoneSegmentOffset = oldTombstoneOffset + numSegmentsFullyConsumed;

  if (lastSegmentIndex == newTombstoneSegmentOffset) {
    uint256 numOrdersWithinLastSegmentFullyConsumed = numOrdersFullyConsumed % NUM_SLOTS_PER_SEGMENT;
    lastSegment = lastSegment.clearUpToSlot(numOrdersWithinLastSegmentFullyConsumed);

    if (uint256(lastSegment) == 0) {
      newTombstoneSegmentOffset = lastSegmentIndex + 1;
    } else {
      uint256 compressedStatus = lastSegment.compressedStatus();
      require(compressedStatus != 2, PriceSegmentHasAHole());
      if (compressedStatus > 3) {
        require(lastSegmentIndex == segments.length - 1, PriceSegmentHasAHole());
      }

      segments[lastSegmentIndex] = lastSegment;
    }
  } else {
    require(lastSegmentIndex + 1 == newTombstoneSegmentOffset,
    ↪  IncorrectDiffBetweenLastSegmentAndTheNewTombstoneOffset());
  }
```

```
    for (uint256 i = oldTombstoneOffset; i < newTombstoneSegmentOffset; ++i) {
      delete segments[i];
    }

    if (newTombstoneSegmentOffset != oldTombstoneOffset) {
      tree.edit(bestPrice, uint56(newTombstoneSegmentOffset));
    }

    if (newTombstoneSegmentOffset >= segments.length) {
      tree.remove(bestPrice);
    }
  }
```

```
function _updateSegmentsAndTree(/*...*/) private {
  // ...
  if (lastSegmentIndex == newTombstoneSegmentOffset) {
    // ...
    if (uint256(lastSegment) == 0) {
      // ...
      newTombstoneSegmentOffset = lastSegmentIndex + 1;
      // Branch A
    } else {
      // ...
      segments[lastSegmentIndex] = lastSegment;
      // Branch B
    }
  } else {
    require(lastSegmentIndex + 1 == newTombstoneSegmentOffset,
    ↪  IncorrectDiffBetweenLastSegmentAndTheNewTombstoneOffset());
    // Branch C
  }

  // ...
}
```

Method	Inv 1.	Inv 2.	Inv 3.	Inv 4.	Inv 5.	Inv 6.	Inv 7.	Inv. 8	Inv. 9	Inv 10.
matching orders (patch above) Branch A	*	✓	N/A	✓	✓	✓	✓	✓	✓	✓
matching orders (patch above) Branch B	**	****	N/A	✓	✓	✓	N/A	✓	✓	✓
matching orders (patch above) Branch C	***	✓	N/A	✓	✓	✓	✓	✓	✓	✓

- * (Branch A, Inv 1.): If the a hole was about to get introduced it will be ignored. The final price segment would not end up having a hole.
- ** (Branch C, Inv 1.): If the a hole was about to get introduced it will be ignored. The final price segment would not end up having a hole.
- *** (Branch B, Inv 1.): If the a hole was about to get introduced it will be ignored the transaction would revert with `PriceSegmentHasAHole()`. One still need to check the upstream flow in

  `_consumePriceSegments` → `loops` → `_parseOrder` to make sure one would not hit error. The upside is the new approach would prevent price segmements having holes and in case of an error one needs to check the upstream logic.

  1. One can prove that `_parseOrder` preserves the compressed status (ie the shape) of a segment and thus througout the loops in `_consumePriceSegments` up to the point were one enters `_updateSegmentsAndTree` the compressed status (the shape) if the price segment stays the same. Thus the shapes used for invariants 1, 2, 5, 6, 8, 9 are not modified while parsing through the price segment slots.
  2. In this whole flow it is important that once one partially matches a slot, the loops should terminate and the paramters `numOrdersFullyConsumed`, `segment` and `lastConsumedSegmentIndex` should not change after this point on (!!).

     Checking and proving the above points 1. and 2. should prove that **invariant 1** is satisfied for the matching order flow (before and after jumping in `_updateSegmentsAndTree`).

- **** (Branch B, Inv 2.): `_parseOrder` needs to be verified to not break this invariant. It has already been checked that the slot being parsed in `_parseOrder` has a non-zero `orderId` before `_parseOrder` gets called. So we just need to make sure the normalised quantity does not end up being 0 for this slot (✓ proof below).
- Case 1. budget-constrained buy market orders:

$$(0 < q_n) \wedge \left(c_r < \left\lfloor \frac{q_n \cdot \Delta q \cdot p_b}{10^d} \right\rfloor\right) \Rightarrow q_n - \left\lfloor \frac{\left\lfloor \frac{c_r \cdot 10^d}{p_b} \right\rfloor}{\Delta q} \right\rfloor > 0$$

  One can prove:

$$0 \leq \left(\frac{10^d}{\Delta q \cdot p_b}\right)\epsilon_0 + \left(\frac{1}{\Delta q}\right)\epsilon_1 + \epsilon_2 < q_n - \left\lfloor \frac{\left\lfloor \frac{c_r \cdot 10^d}{p_b} \right\rfloor}{\Delta q} \right\rfloor$$

  where $\epsilon_0, \epsilon_1, \epsilon_2 \in [0, 1)$.
- Case 2. all other cases: This is obvious based on `if/else` branch conditions.
- **Canceling orders:** Applying the recommendation in "unsafe node check in `_cancelOrdersSide`" is recommended.
- **Summary:** Apply the following patch. This would cover the changes required for:
- Finding 1.
- Finding 2.
- Finding 3.
- Finding 5 (make sure price ticks are multiple of 100).
- Finding 6 (patch above plus some improvements).
- Finding 7 (order cancelation flow checks, order addition checks needs another patch).

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index 31b5486..cd40406 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -108,6 +108,12 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
     error SellMustUseExactQuantity();
     error MakerPriceMustMatch(uint256 expectedPrice, uint256 actualPrice);

+    // Errors related to price segment invariants being broken
+    error PriceSegmentHasAHole();
+    error IncorrectDiffBetweenLastSegmentAndTheNewTombstoneOffset();
+    error LastPriceSegmentCannotBeEmpty();
+    error OnlyTheTombstoneSegmentCanStartWithEmptySlots();
+
     enum OrderSide {
       Buy,
       Sell
  @@ -158,7 +164,7 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
       uint256 currentCost;
       uint256 fee;
       uint256 numberOfOrdersTraded;
-      bool isBudgetConstrainedBuyingLimitReached;
+      bool stopConsumingMoreOrders;
     }

     // A helper type for a view function to return orders. Packing doesn't matter
  @@ -681,7 +687,7 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
     }

     function _shouldTakeMoreFromOrderBook(TakeFromOrderBookParams memory params) internal pure returns (bool) {
-      if (params.isBudgetConstrainedBuyingLimitReached) {
+      if (params.stopConsumingMoreOrders) {
         return false;
       }
```

```
@@ -773,7 +779,10 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
      uint256 lastConsumedSegmentIndex;
      uint16 devFeeTakerBps = _devFeeTakerBps;

-     for (uint256 i = node.tombstoneSegmentOffset; i < segments.length &&
↪  _shouldTakeMoreFromOrderBook(params); ++i) {
+     uint256 segmentsLength = segments.length;
+     uint256 tombstoneSegmentOffset = node.tombstoneSegmentOffset;
+
+     for (uint256 i = tombstoneSegmentOffset; i < segmentsLength && _shouldTakeMoreFromOrderBook(params);
↪  ++i) {
        lastConsumedSegmentIndex = i;
        segment = segments[i];

@@ -786,15 +795,67 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
          bytes32 remainingSegment = segment.getRemainingSegment(slotIndex);
          if (remainingSegment == 0) {
            // Should only hit this branch on the last segment
+
+           // but if for some reason we hit this branch before
+           // the last segment we add the count of slots skipped
+           if (i < segmentsLength - 1) {
+             // This branch which should not be reachable
+             revert PriceSegmentHasAHole();
+           } else {
+             // we are processing the last price segment in this branch
+             // the possibilities are (o: an empty slot, s: slotIndex):
+             // o    o    o(s) - BAD
+             // o    o(s) ...  - skip incrementing `numOrdersFullyConsumed`
+             // o(s) ...  ...  - skip incrementing `numOrdersFullyConsumed`
+             if (slotIndex == 0) {
+               // This branch which should not be reachable
+               // the last segment in the active area of the price segment
+               // only has empty slots
+               // o    o    o(s) - BAD
+               revert LastPriceSegmentCannotBeEmpty();
+             }
+           }
+

            break;
          }

          bytes10 slot = segment.getSlot(slotIndex);
          uint48 orderId = slot.getOrderId();
          if (orderId == 0) {
-           // Skip this order in the segment as it's been deleted
-           ++numOrdersFullyConsumed;
-           continue;
+           // we should only hit this branch at the very beginning of the
+           // active area of the price segment pointed to by the `tombstoneSegmentOffset`
+           // Possible configurations of this segment should be of the form:
+           // (o: an empty slot, x: a non empty slot, s: slotIndex, ot = tombstoneSegmentOffset)
+           // x o    o(s)  <-- (i === ot) - allowed
+           // x o(s) o     <-- (i === ot) - allowed
+           // x x    o(s)  <-- (i === ot) - allowed
+           //
+           // x o    o(s)  <-- (i =/= ot) - BAD
+           // x o(s) o     <-- (i =/= ot) - BAD
+           // x x    o(s)  <-- (i =/= ot) - BAD
+           // x o(s) x     <-- (i  >= ot) - BAD
+
+           if (i == tombstoneSegmentOffset) {
+             // x o    o(s)  <-- (i === ot) - allowed
+             // x o(s) o     <-- (i === ot) - allowed
+             // x x    o(s)  <-- (i === ot) - allowed
+             //
+             // @dev we are allowing this potentially unreachable state below to pass through
+             //      and just increment `numOrdersFullyConsumed`
+             // x o(s) x     <-- (i  == ot) - BAD

+             // Skip this order in the segment as it's been deleted
+             ++numOrdersFullyConsumed;
+             continue;
+           } else {
+             // this branch should be unreachable if all invariants are satisfied
```

```
+              // x o    o(s)  <-- (i =/= ot) - BAD
+              // x o(s) o     <-- (i =/= ot) - BAD
+              // x x    o(s)  <-- (i =/= ot) - BAD
+              // x o(s) x     <-- (i =/= ot) - BAD
+
+              revert OnlyTheTombstoneSegmentCanStartWithEmptySlots();
+            }
          }

          uint256 ordersConsumed = 0;
  @@ -836,13 +897,13 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
        uint256 remainingBudget = params.maxCost - params.currentCost;

        if (fullOrderCost > remainingBudget) {
+         // // Can't afford a full order, stop here as the slot will be potentially partially touched.
+         params.stopConsumingMoreOrders = true;
          // Check if we can afford partial order
          uint256 maxAffordableQuantity = (remainingBudget * BASE_DECIMAL_DIVISOR) / params.bestPrice;
          // Floor to QUANTITY_TICK to ensure proper alignment
          maxAffordableQuantity = (maxAffordableQuantity / QUANTITY_TICK) * QUANTITY_TICK;
          if (maxAffordableQuantity == 0) {
-           // Can't afford any of this order, stop here
-           params.isBudgetConstrainedBuyingLimitReached = true;
            return (
              segment,
              0, // ordersConsumed
  @@ -880,6 +941,7 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
          slot = slot.setQuantity(uint88(quantityL3 - quantityRemaining));
          segment = segment.setSlot(slotIndex, slot);
          quantityNFTClaimable = quantityRemaining;
+         params.stopConsumingMoreOrders = true;
        }

        // at this point one should be able to prove that:
  @@ -915,39 +977,126 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
      uint256 lastSegmentIndex,
      uint256 numOrdersFullyConsumed
    ) private {
+     // `segments.length > lastSegmentIndex >= oldTombstoneOffset`
+     // `numOrdersFullyConsumed` should be the number of order slots fully consumed
+     // yet not modified but the cursor passed over it to the next order slot.
+
+     // Assumptions:
+     //    1. `numSegmentsFullyConsumed` is calculated correctly, it should be the number of
+     //       slots fully consumed or skipped if the slot was EMPTY (0)
+     //    2. `lastSegmentIndex` points to the last segment consumed
+     //    3. `lastSegment` is the segment pointed to by `lastSegmentIndex`
+
+     // [CHECK LIST] One should enforce the following for `lastSegment` (the last segment consumed):
+     // (o: an EMPTY slot, x: a non-EMPTY slot)
+     //
+     // o o o - last consumed segment - ALLOWED (only if the price node gets removed from the tree)
+     // o o x - last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to this slot too)
+     // o x o - last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to this slot too)
+     // o x x - last consumed segment - ALLOWED (check `seg.len - 1` point to this slot too)
+     // x o o - last consumed segment - ALLOWED (check `ot` point to this slot too)
+     // x o x - last consumed segment - CATCH - no hole in the middle allowed
+     // x x o - last consumed segment - ALLOWED (check `ot` point to this slot too)
+     // x x x - last consumed segment - ALLOWED
+
      uint256 numSegmentsFullyConsumed = numOrdersFullyConsumed / NUM_SLOTS_PER_SEGMENT;
-     uint256 numOrdersWithinLastSegmentFullyConsumed = numOrdersFullyConsumed % NUM_SLOTS_PER_SEGMENT;
+     uint256 oldTombstoneOffset = bestPriceNode.tombstoneSegmentOffset;
+     uint256 newTombstoneSegmentOffset = oldTombstoneOffset + numSegmentsFullyConsumed;

-     uint56 oldTombstoneOffset = bestPriceNode.tombstoneSegmentOffset;
-     if (numSegmentsFullyConsumed != 0) {
-       // Cleanup segments to get a gas refund
-       for (uint256 i = 0; i < numSegmentsFullyConsumed; ++i) {
-         delete segments[oldTombstoneOffset + i];
-       }
+     if (lastSegmentIndex == newTombstoneSegmentOffset) {
+       // At this branch we know `segments.length > lastSegmentIndex >= newTombstoneSegmentOffset`
+       uint256 numOrdersWithinLastSegmentFullyConsumed = numOrdersFullyConsumed % NUM_SLOTS_PER_SEGMENT;
+       lastSegment = lastSegment.clearUpToSlot(numOrdersWithinLastSegmentFullyConsumed);
```

```
-        uint256 newTombstoneSegmentOffset = oldTombstoneOffset + numSegmentsFullyConsumed;
-        tree.edit(bestPrice, uint56(newTombstoneSegmentOffset));
+        if (uint256(lastSegment) == 0) {
+          // `lastSegment` has this property/assumption that either:
+          // A1. An other was partially matched in this segment or
+          // A2. We had reached the last segment in the active area of the price segments which
+          //     has some zero slots (`if (remainingSegment == 0)` branch in `_consumePriceSegments`)
+          //
+          // Moreover, there is an assumption here that if `lastSegment` that was touched during the
+          // matching/filling process ends up being `0` then that would mean, there are no orders
+          // left in the price segments since A1 would not be possible and so for A2:
+          // 1. any order slot before this segment should have been fully filled/matched and thus
+          //    should have been zeroed out in the previous `if (numSegmentsFullyConsumed != 0)`
+          //    block.
+          // 2. There could not be any non-zero order slot past this segment since otherwise,
+          //    that would indicate there would have been holes in the price segments previously
+          //    breaking (Inv 1.)
+          newTombstoneSegmentOffset = lastSegmentIndex + 1;
+          // so we would have:
+          // `segments.length >= newTombstoneSegmentOffset == lastSegmentIndex + 1`
+          // `newTombstoneSegmentOffset == lastSegmentIndex + 1 > oldTombstoneOffset`
+          // thus below `lastSegment` shuold get deleted
+
+          // o o o - last consumed segment - ALLOWED - it will get deleted and the new tombstone offset
+          //                                           points to the following segment
+
+          // The checks from the [CHECK LIST] are not required since if one hits this branch and continue
+          // without reverting, the last segment consumed would be removed from the price segments
+          // and the new tombstone offset would point to the following segment.
+        } else {
+          // At this branch we know `segments.length > lastSegmentIndex >= newTombstoneSegmentOffset`
+          // and `lastSegment` is non-zero, so at the end `bestPrice` would not get removed from the `tree`

-        // Consumed all orders at this price level, so remove it from the tree
-        if (newTombstoneSegmentOffset == segments.length) {
-          tree.remove(bestPrice);
-        }
-      }

-      bool lastSegmentDeleted = lastSegmentIndex < (oldTombstoneOffset + numSegmentsFullyConsumed);
-      if (!lastSegmentDeleted) {
-        if (numOrdersWithinLastSegmentFullyConsumed != 0) {
-          for (uint256 i; i < numOrdersWithinLastSegmentFullyConsumed; ++i) {
-            lastSegment = lastSegment.clearSlot(i);
+          // s2 s1 s0 : cs(bits) : lastSegment <-- (lastSegmentIndex == newTombstoneSegmentOffset)
+          // --------------------------------(si: the `i`th slot, cs: compressed status)-------
+          //  o  o  o :      111 : last consumed segment - ALLOWED (only if the price node gets removed from
↪  the tree)
+          //  o  o  x :      110 : last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to this
↪  slot too)
+          //  o  x  o :      101 : last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to this
↪  slot too)
+          //  o  x  x :      100 : last consumed segment - ALLOWED (check `seg.len - 1` point to this slot too)
+          //  x  o  o :      011 : last consumed segment - ALLOWED (check `ot` point to this slot too)
+          //  x  o  x :      010 : last consumed segment - CATCH - no hole in the middle allowed
+          //  x  x  o :      001 : last consumed segment - ALLOWED (check `ot` point to this slot too)
+          //  x  x  x :      000 : last consumed segment - ALLOWED
+
+          uint256 compressedStatus = lastSegment.compressedStatus();
+
+          //  o  o  o : 111 : one would not enter this branch for this case
+
+          //  x  o  x : 010 : last consumed segment - CATCH - no hole in the middle allowed
+          require(compressedStatus != 2, PriceSegmentHasAHole());
+
+          if (compressedStatus > 3) {
+            //  o  x  x : 100 : last consumed segment - ALLOWED (check `seg.len - 1` point to this slot too)
+            //  o  x  o : 101 : last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to this slot
↪  too)
+            //  o  o  x : 110 : last consumed segment - ALLOWED (check `ot` & `seg.len - 1` point to this slot
↪  too)
+            require(lastSegmentIndex == segments.length - 1, PriceSegmentHasAHole());
+          }
-        }

-        if (uint256(lastSegment) == 0) {
```

```
-          // All orders in the last segment are consumed, delete from tree
-          tree.remove(bestPrice);
-          // Don't pop segment for gas efficiency, instead just write zeros
+          segments[lastSegmentIndex] = lastSegment;
         }
-        segments[lastSegmentIndex] = lastSegment;
+      } else {
+        // if `lastSegmentIndex =/= newTombstoneSegmentOffset` that would mean the matching process had ended
↪  up
+        // with ( - dashes the slot can be 0 or not, c means that it was consumed):
+        // ...
+        // x(c) -(c) -(c) : lastSegmentIndex
+        // where in the last segment, the last slot was fully filled and thus we would end up with:
+        // ...
+        // x(c) -(c) -(c) : lastSegmentIndex
+        //   -  -    -    : newTombstoneSegmentOffset
+        // So these values can only be off by `1`
+        require(lastSegmentIndex + 1 == newTombstoneSegmentOffset,
↪  IncorrectDiffBetweenLastSegmentAndTheNewTombstoneOffset());
+
+        // The checks from the [CHECK LIST] are not required since if one hits this branch and continue
+        // without reverting, the last segment consumed would be removed from the price segments
+        // and the new tombstone offset would point to the following segment.
+      }
+
+      // Cleanup segments to get a gas refund
+      for (uint256 i = oldTombstoneOffset; i < newTombstoneSegmentOffset; ++i) {
+        delete segments[i];
+      }
+
+      if (newTombstoneSegmentOffset != oldTombstoneOffset) {
+        tree.edit(bestPrice, uint56(newTombstoneSegmentOffset));
+      }
+
+      // below one can add an extra branch for `newTombstoneSegmentOffset > segments.length`
+      // and possibly revert since that would mean some invariant was broken.
+      if (newTombstoneSegmentOffset >= segments.length) {
+        tree.remove(bestPrice);
       }
     }

  @@ -1000,7 +1149,7 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
       bytes32[] storage segments,
       BokkyPooBahsRedBlackTreeLibrary.Tree storage tree
     ) private returns (uint88 quantity) {
-      BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree.getNodeUnsafe(price);
+      BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree.getNode(price);
       uint256 tombstoneSegmentOffset = node.tombstoneSegmentOffset;

       (uint256 segmentIndex, uint256 slotIndex) = _find(segments, tombstoneSegmentOffset, segments.length,
       ↪  orderId);
  @@ -1058,7 +1207,7 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
         currentCost: 0,
         fee: 0,
         numberOfOrdersTraded: 0,
-        isBudgetConstrainedBuyingLimitReached: false
+        stopConsumingMoreOrders: false
       });

       _takeFromOrderBook(params, orderIdsPool, quantitiesPool, pricesPool);
  @@ -1105,7 +1254,7 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
         params.currentCost = 0;
         params.fee = 0;
         params.numberOfOrdersTraded = 0;
-        params.isBudgetConstrainedBuyingLimitReached = false;
+        params.stopConsumingMoreOrders = false;
       }

       _takeFromOrderBook(params, orderIdsPool, quantitiesPool, pricesPool);
  @@ -1409,6 +1558,17 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
           bytes32 currentSegment = segments[currentSegmentIndex];
           bytes10 nextSlot = currentOrNextSegment.getSlot(nextSlotIndex);
           if (nextSlot == 0) {
+            // When reaching a hole, make sure it has iterated to the last order.
+
+            // Must iterate to the last seg
```

```
+            require(nextSegmentIndex == segments.length - 1, PriceSegmentHasAHole());
+
+            // nextSlot and subsequent slots must be empty
+            require(currentOrNextSegment.getRemainingSegment(nextSlotIndex) == 0, PriceSegmentHasAHole());
+
+            // The last non-empty order must be in `seg[seg.len-1]` (`seg.len-1` cannot point to an empty
↪  segment).
+            require(nextSlotIndex > 0, LastPriceSegmentCannotBeEmpty());
+
             // There are no more orders left, reset back to the currently iterated order as the last
             nextSegmentIndex = currentSegmentIndex;
             nextSlotIndex = currentSlotIndex;
  diff --git a/contracts/libraries/SegmentLibrary.sol b/contracts/libraries/SegmentLibrary.sol
  index d884717..e7c91a2 100644
-  -- a/contracts/libraries/SegmentLibrary.sol
+  ++ b/contracts/libraries/SegmentLibrary.sol
  @@ -43,6 +43,39 @@ library SegmentLibrary {
       return bytes10(uint80((uint256(segment) >> (slotIndex * SEGMENT_SIZE)) & SEGMENT_MASK));
     }

+    /// @notice Compresses the segment into 3 indicator bits
+    /// @param segment The segment to read from
+    /// @return status with only 3 bits sets each indicating whether the corresponding
+    //                 slot in the segment is EMPTY or not.
+    function compressedStatus(bytes32 segment) internal pure returns (uint256 status) {
+      uint256 MASK = SEGMENT_MASK;
+      uint256 SIZE = SEGMENT_SIZE;
+
+      assembly ("memory-safe") {
+        // s2 s1 s0 : segment
+        status := iszero(and(segment, MASK))
+        segment := shr(SIZE, segment)
+
+        //  0 s2 s1 : segment
+        status := or(
+          shl(1, iszero(and(segment, MASK))),
+          status
+        )
+        segment := shr(SIZE, segment)
+
+        //  0  0 s2 : segment
+        status := or(
+          shl(2, iszero(and(segment, MASK))), // still mask in case segment has dirty uppper bits
+          status
+        )
+
+        // b2 b1 b0 : status bits (it would only have 3 set bits)
+        // b2       : is 1 if s2 the 3rd slot of original segment was EMPTY
+        //    b1    : is 1 if s1 the 2nd slot of original segment was EMPTY
+        //       b0 : is 1 if s0 the 1st slot of original segment was EMPTY
+      }
+    }
+
     /// @notice Checks if the segment only contains data in the specified slot
     /// @dev Returns true if all other slots in the segment are empty (zero)
     /// @param segment The segment to check
  @@ -60,6 +93,19 @@ library SegmentLibrary {
       return bytes32(uint256(segment) & ~(uint256(SEGMENT_MASK) << (slotIndex * SEGMENT_SIZE)));
     }

+    /// @notice Clears slots up to the specified index, setting all its bits to zero
+    /// @param segment The segment to modify
+    /// @param slotIndex The index of the slot to clear up to (0, 1, or 2)
+    /// @return newSegment The updated segment with the right most slots cleared up to `slotIndex`
+    function clearUpToSlot(bytes32 segment, uint256 slotIndex) internal pure returns (bytes32 newSegment) {
+      if (slotIndex == 0) {
+        return segment;
+      }
+
+      uint256 shiftAmount = slotIndex * SEGMENT_SIZE;
+      return bytes32((uint256(segment) >> (shiftAmount)) << shiftAmount);
+    }
+
     /// @notice Returns the remaining segment data after the specified slot index
     /// @dev Shifts the segment right to remove all slots up to and including the specified index.
     ///      Useful for iterating through higher-indexed slots or processing remaining data.
```

```
  diff --git a/contracts/libraries/SonicOrderBookLibrary.sol b/contracts/libraries/SonicOrderBookLibrary.sol
  index de229f7..08cd374 100644
-  -- a/contracts/libraries/SonicOrderBookLibrary.sol
+  ++ b/contracts/libraries/SonicOrderBookLibrary.sol
  @@ -31,6 +31,7 @@ library SonicOrderBookLibrary {
     error DevFeeTooHigh();
     error DevFeeNotSet();
     error MaxOrdersNotMultipleOfOrdersInSegment();
+    error PriceTickShouldBeAMutipleOf100();

     uint256 private constant SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME = 7 days;

  @@ -57,6 +58,7 @@ library SonicOrderBookLibrary {

         require(existingTick == 0 || existingTick == newTick, PriceTickCannotBeChanged());
         require(newTick != 0, PriceTickCannotBeZero());
+        require(newTick % 100 == 0, PriceTickShouldBeAMutipleOf100());

         SeasonData memory seasonData = nft.getSeasonData(Season(tokenId));
         if (seasonData.startTime != 0 && tokenIdInfos[i].isTradeable) {
```

Paintswap: Fixed in PR 106.

Cantina Managed: Fix verified.

3.3 Medium Risk

3.3.1 Unsafe node check in `_cancelOrdersSide`

Severity: Medium Risk

Context: SonicOrderBook.sol#L1003

Description: The price `node` queries from the price `tree` uses the `getNodeUnsafe` which omits checking whether the price `node` belongs to the `tree`. Ommitting this check was added to save some gas during the cancellation process but creates big risk in terms of making assumptions about the structure of the price segments. The assumption is that if the price `node` did not belong the the `tree` then the `node`'s tombstone offset should be equal to greater than the price segments length:

$$o_t \geq \texttt{seg.len}$$

Thus when one calls `_find` the order should not be found and `type(uint256).max` should be returned for the segment and slot indexes, and so instead one would hit the custom error `OrderNotFound(orderId))` below:

```
require(segmentIndex != type(uint).max, OrderNotFound(orderId));
```

One is creating assumptions about distant parts of the codebase using an optimistic approach about supposed invariants.

Recommendation: This is a critical flow for orderbook and minimising assumptions to reduce the risk of accounting imbalances would be the recommended suggestion. Thus it would be best to revert back to the original logic of querying the `node` in a safe way:

```
BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree.getNode(price);
```

Thus would make it harder to cancel a potentially malicious non-existent order and steal funds from the orderbook and it is a more defensive approach about safeguarding the contract balances.

Paintswap: Fixed in PR 106.

Cantina Managed: Fix verified.

3.3.2 Add invariant checks to the `_consumePriceSegments` and make it less prone to malfunction

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: There are currently two main spots in `_consumePriceSegments` which makes certain assumptions about the structure of the price segments. Mainly:

1. Only the beginning of the active area of the price segments can have empty slots.

   which is assumed in the snippet below:

   ```
   bytes10 slot = segment.getSlot(slotIndex);
   uint48 orderId = slot.getOrderId();
   if (orderId == 0) {
      // Skip this order in the segment as it's been deleted
      ++numOrdersFullyConsumed;
      continue;
   }
   ```

2. Only the end the of the active area of the price segments can have empty slots.

   which is assumed in the snippet below:

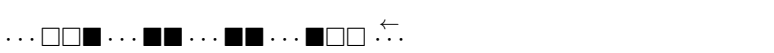
   ```
   bytes32 remainingSegment = segment.getRemainingSegment(slotIndex);
   if (remainingSegment == 0) {
      // Should only hit this branch on the last segment
      break;
   }
   ```

3. There are no holes / empty slots in the middle. Configurations like below should not be present:

    invariant is broken

   In a presence of holes `numOrdersFullyConsumed` would not get updated correctly which eventually in `_updateSegmentsAndTree` would cause the price segment and tree get updated in correctly in the storage.

i.e, if we lined up all the slots of the active area of the price segment it would look like:



Proving the above invariant as well as showing that the order ids of the non-empty slots ■ increase monotonically involves many edge cases from different lines of the order book contract. But this these are very important invariants and in case they get broken, many of the endpoints and storage updates would not act as intended or get modified correctly.

Recommendation: It would be best to add invariant checks to the `_consumePriceSegments` and make it less prone to malfunction in the presence of holes in price segments:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index 31b5486..13064b8 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -108,6 +108,10 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
     error SellMustUseExactQuantity();
     error MakerPriceMustMatch(uint256 expectedPrice, uint256 actualPrice);

+    // Errors that indicate invariants are broken
+    error LastPriceSegmentCannotBeEmpty();
+    error OnlyTheTombstoneSegmentCanStartWithEmptySlots();
+
     enum OrderSide {
       Buy,
       Sell
  @@ -773,7 +777,9 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
       uint256 lastConsumedSegmentIndex;
       uint16 devFeeTakerBps = _devFeeTakerBps;
```

```
-       for (uint256 i = node.tombstoneSegmentOffset; i < segments.length &&
↪  _shouldTakeMoreFromOrderBook(params); ++i) {
+       uint256 segmentsLength = segments.length;
+       uint256 tombstoneSegmentOffset = node.tombstoneSegmentOffset;
+       for (uint256 i = tombstoneSegmentOffset; i < segmentsLength && _shouldTakeMoreFromOrderBook(params);
↪  ++i) {
          lastConsumedSegmentIndex = i;
          segment = segments[i];

  @@ -786,15 +792,74 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
            bytes32 remainingSegment = segment.getRemainingSegment(slotIndex);
            if (remainingSegment == 0) {
              // Should only hit this branch on the last segment
+             // but if for some reason we hit this branch before
+             // the last segment we add the count of slots skipped
+             if (i < segmentsLength - 1) {
+               // This branch which should not be reachable
+               // But if it does hit we can be proactive and skip the remaining slots
+               // but still add them to `numOrdersFullyConsumed`
+               // `s` is the `slotIndex` and `o` represent an empty slot:
+               // o    o    o(s) - +3
+               // o    o(s) ...  - +2
+               // o(s) ...  ...  - +1
+               //
+               // @dev one can also decide to REVERT below instead of accounting for the
+               //      holes (empty slots) in this scenario
+               numOrdersFullyConsumed += NUM_SLOTS_PER_SEGMENT - slotIndex;
+             } else {
+               // we are processing the last price segment in this branch
+               // the possibilities are (o: an empty slot, s: slotIndex):
+               // o    o    o(s) - BAD
+               // o    o(s) ...  - skip incrementing `numOrdersFullyConsumed`
+               // o(s) ...  ...  - skip incrementing `numOrdersFullyConsumed`
+               if (slotIndex == 0) {
+                 // This branch which should not be reachable
+                 // the last segment in the active area of the price segment
+                 // only has empty slots
+                 // o    o    o(s) - BAD
+                 revert LastPriceSegmentCannotBeEmpty();
+               }
+             }
              break;
            }

            bytes10 slot = segment.getSlot(slotIndex);
            uint48 orderId = slot.getOrderId();
            if (orderId == 0) {
-             // Skip this order in the segment as it's been deleted
-             ++numOrdersFullyConsumed;
-             continue;
+             // we should only hit this branch at the very beginning of the
+             // active area of the price segment pointed to by the `tombstoneSegmentOffset`
+             // Possible configurations of this segment should be of the form:
+             // (o: an empty slot, x: a non empty slot, s: slotIndex, ot = tombstoneSegmentOffset)
+             // x o    o(s)  <-- (i === ot) - allowed
+             // x o(s) o     <-- (i === ot) - allowed
+             // x x    o(s)  <-- (i === ot) - allowed
+             //
+             // x o    o(s)  <-- (i =/= ot) - BAD
+             // x o(s) o     <-- (i =/= ot) - BAD
+             // x x    o(s)  <-- (i =/= ot) - BAD
+             // x o(s) x     <-- (i  >= ot) - BAD
+
+             if (i == tombstoneSegmentOffset) {
+               // x o    o(s)  <-- (i === ot) - allowed
+               // x o(s) o     <-- (i === ot) - allowed
+               // x x    o(s)  <-- (i === ot) - allowed
+               //
+               // @dev we are allowing this potentially unreachable state below to pass through
+               //      and just increment `numOrdersFullyConsumed`
+               // x o(s) x     <-- (i  == ot) - BAD

+               // Skip this order in the segment as it's been deleted
+               ++numOrdersFullyConsumed;
+               continue;
```

```
+            } else {
+              // this branch should be unreachable if all invariants are satisfied
+              // x o    o(s)  <-- (i =/= ot) - BAD
+              // x o(s) o     <-- (i =/= ot) - BAD
+              // x x    o(s)  <-- (i =/= ot) - BAD
+              // x o(s) x     <-- (i =/= ot) - BAD
+
+              revert OnlyTheTombstoneSegmentCanStartWithEmptySlots();
+            }
           }

           uint256 ordersConsumed = 0;
```

In the branch where on is performing:

```
numOrdersFullyConsumed += NUM_SLOTS_PER_SEGMENT - slotIndex;
```

One can instead revert, since it would indicate the precense of holes. But the above make `_consumePriceSegments` hole-resistant (ie, in the presence of holes this function would not get broken and act as expected, although the other flows make suffer from the holes such as `_find` or `_cancelOrder`, ...). Another options would be to instead `revert` here:

```
revert PriceSegmentHasAHole();
```

Paintswap: Fixed in PR 106.

Cantina Managed: Fix verified.

3.3.3 Quote tokens received by the orderbook versus to-be-claimed quote tokens by sellers

Severity: Medium Risk

Context: SonicOrderBook.sol#L896-L897

Description: When a sell order is partially matched a few times the amount of quote tokens received by the orderbook would be:

$$\sum_j \left\lfloor \frac{q_j \cdot p_o}{10^d} \right\rfloor$$

where j iterates over the partial match events of one specific order o and p_o is the price for this order, q_j are the partial quantities filled. d is the decimal of the `NFT` contract which in this case would be 18.

When the user wants to claim these partial matches in `_claimTokens` and `_calculateClaimableTokens` the amount to be claimed and sent to the `maker` and the `dev` would be:

$$\left\lfloor \frac{(\sum_j q_j) \cdot p_o}{10^d} \right\rfloor$$

But we have:

$$\left\lfloor \frac{(\sum_j q_j) \cdot p_o}{10^d} \right\rfloor \geq \sum_j \left\lfloor \frac{q_j \cdot p_o}{10^d} \right\rfloor$$

Thus if these divisions are not exact, the orderbook would not have enough quote tokens to compensate for the claims.

Recommendation: Either one needs to make sure the divisions are exact by setting restrictions on the `priceTicks` and quantity ticks to make sure their product would be division by 10^d. Moreover one needs to throughout the codebase make sure that the invariant of:

1. Prices being divisible by `priceTick`.
2. Quantities being divisible by quantity tick.

to be satisfied. For the above apply this patch:

```
  diff --git a/contracts/libraries/SonicOrderBookLibrary.sol b/contracts/libraries/SonicOrderBookLibrary.sol
  index de229f7..08cd374 100644
-  -- a/contracts/libraries/SonicOrderBookLibrary.sol
+  ++ b/contracts/libraries/SonicOrderBookLibrary.sol
  @@ -31,6 +31,7 @@ library SonicOrderBookLibrary {
     error DevFeeTooHigh();
     error DevFeeNotSet();
     error MaxOrdersNotMultipleOfOrdersInSegment();
+    error PriceTickShouldBeAMultipleOf100();

     uint256 private constant SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME = 7 days;

  @@ -57,6 +58,7 @@ library SonicOrderBookLibrary {

         require(existingTick == 0 || existingTick == newTick, PriceTickCannotBeChanged());
         require(newTick != 0, PriceTickCannotBeZero());
+        require(newTick % 100 == 0, PriceTickShouldBeAMultipleOf100());

         SeasonData memory seasonData = nft.getSeasonData(Season(tokenId));
         if (seasonData.startTime != 0 && tokenIdInfos[i].isTradeable) {
```

Another solution would be to simply overestimate the case (this would not require checking for the above conditions at least for cost related issues). Overestimating would guarantee that for partially matching a specific order multiple times we would have:

$$\left\lceil \frac{(\sum_j q_j) \cdot p_o}{10^d} \right\rceil \leq \sum_j \left\lceil \frac{q_j \cdot p_o}{10^d} \right\rceil$$

i.e, in this case the orderbook should have enough quote tokens to cover the cost of a maker whose sell orders get partially filled a few times.

So in summary:

- Overestimate the cost for matching into sell orders.
- Underestimate the cost for matching into buy orders.
- Overestimate the cost for placing buy limit orders.

Paintswap: Fixed in PR 106.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Add invariant checks when adding and canceling orders

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The protocol modifies segments when adding/canceling/matching orders.

When matching orders, "Add invariant checks to the `_consumePriceSegments` and make it less prone to malfunction" checks segments in `_consumePriceSegments()` to ensure that the segment state follows the invariant, and "None" also makes some assumption checks when modifying segments in `_updateSegmentsAndTree()`.

For adding/canceling orders, some invariant checks need to be added to ensure that the segment state is as expected.

Below are all cases of slots in the segment:

```
(1) o o o  BAD
(2) o o x  ALLOW, holes Must be tail
(3) o x o  ALLOW, holes Must be head&tail
(4) x o o  ALLOW, holes Must be head
(5) o x x  ALLOW, holes Must be tail
(6) x o x  BAD
(7) x x o  ALLOW, holes Must be head
(8) x x x  ALLOW, No holes
```

Recommendation: For order additions:

```
  @@ -1204,7 +1207,12 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
           // Are there are free slots in this segment
           for (uint256 slotIndex = 0; slotIndex < NUM_SLOTS_PER_SEGMENT; ++slotIndex) {
             bytes32 remainingSegment = lastSegment.getRemainingSegment(slotIndex);
+            // o o o, o o x, o x x, o x o,
             if (remainingSegment == 0) {
+              // catch o o o
+              require(slotIndex != 0, LastPriceSegmentCannotBeEmpty());
+              // for o x o, must be head
+              if(lastSegment.getSlot(0) == 0) require(lastSegmentIndex == node.tombstoneSegmentOffset,
↪  PriceSegmentHasAHole());
               // Found free slot, so add this order to an existing segment
               bytes10 slot = SlotLibrary.createSlot(orderId, normalizedQuantity);
               bytes32 newSegment = lastSegment.setSlotUnsafe(slotIndex, slot);
  @@ -1212,6 +1220,18 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
             return;
           }
         }
+        // slot2 is x, x o o, x x o, x o x, x x x
+        // head&tail, catch x o x, allow x o o , x x o, x x x
+        if (lastSegmentIndex == node.tombstoneSegmentOffset) {
+          if(lastSegment.getSlot(1) == 0) require(lastSegment.getSlot(0) == 0, PriceSegmentHasAHole());
+        }
+        // tail, must be x, x ,x
+        if (lastSegmentIndex > node.tombstoneSegmentOffset) {
+          require(lastSegment.getSlot(0) != 0 && lastSegment.getSlot(1) != 0, PriceSegmentHasAHole());
+        }
+      } else {
+        // ot is at most equal to seg.len
+        require(lastSegmentIndex + 1 == node.tombstoneSegmentOffset,
↪  IncorrectDiffBetweenLastSegmentAndTombstoneOffset());
       }
     }
```

For order cancellation:

```
  @@ -1410,6 +1428,10 @@ contract SonicOrderBook is UUPSUpgradeable, OwnableUpgradeable, ERC1155Holder, R
           bytes10 nextSlot = currentOrNextSegment.getSlot(nextSlotIndex);
           if (nextSlot == 0) {
             // There are no more orders left, reset back to the currently iterated order as the last
+            // When reaching a hole, make sure it has iterated to the last order.
+            require(nextSegmentIndex == segments.length - 1, PriceSegmentHasAHole()); // Must iterate to the
↪  last seg
+            require(currentOrNextSegment.getRemainingSegment(nextSlotIndex) == 0, PriceSegmentHasAHole()); //
↪  nextSlot and subsequent slots must be empty
+            require(nextSlotIndex > 0, LastPriceSegmentCannotBeEmpty()); // The last non-empty order must be
↪  in seg[seg.len-1](seg.len-1 cannot point to an empty segment).
             nextSegmentIndex = currentSegmentIndex;
             nextSlotIndex = currentSlotIndex;
             break;
```

Paintswap: Fixed in PR 106.

Cantina Managed: Fix verified.

3.4.2 Use `quantityFulfilled` for sell market orders when transferring NFTs

Severity: Informational

Context: SonicOrderBook.sol#L281

Description: `order.quantity` and `quantityFulfilled` are supposed to have the same value in this branch (sell market orders need to use exact quantity).

Recommendation: but to avoid future foot gun it would be best to use `quantityFulfilled` here:

```
_safeTransferNFTsToUs(sender, order.tokenId, quantityFulfilled);
```

This is due to the fact that if the check regarding exact quantities gets modified the actual value traded would be `quantityFulfilled`.

Paintswap: Fixed in PR 106.

Cantina Managed: Fix verified.