

Sonic Airdrop Contracts

Security Review

Cantina Managed review by:

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

July 18, 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 High Risk . . . 4
  - 3.1.1 `safeTransferFrom` and `safeBatchTransferFrom` also transfer tokens after `lockedBurnTime` . . . 4
  - 3.1.2 `placeMarketOrder()` sends incorrect amount of NFTs to users . . . 4
- 3.2 Medium Risk . . . 5
  - 3.2.1 `balanceOfBatch` does not consider the `lockedBurnTime` of the token ids when returning balances . . . 5
  - 3.2.2 Disabling trading can be frontrun . . . 5
  - 3.2.3 The `useExactQuantity` feature is implemented incorrectly . . . 6
  - 3.2.4 The update of remaining quantity, cost and fee in the filling order flow make it hard to reason about invariants . . . 7
- 3.3 Low Risk . . . 18
  - 3.3.1 `setBanned` can be frontrun . . . 18
  - 3.3.2 `isApprovedForAll` does not check for banned operators . . . 19
  - 3.3.3 The distribution branches after placing the market order can be simplified . . . 19
  - 3.3.4 Interactions of the banned users with the orderbook . . . 20
- 3.4 Informational . . . 21
  - 3.4.1 Dangling Comment . . . 21

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 Jun 24th to Jun 26th the Cantina team conducted a review of sonic-airdrop-contracts on commit hash a8e80069. The team identified a total of **11** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	4	3	1
Low Risk	4	2	2
Gas Optimizations	0	0	0
Informational	1	1	0
Total	**11**	**8**	**3**

3 Findings

3.1 High Risk

3.1.1 `safeTransferFrom` and `safeBatchTransferFrom` also transfer tokens after `lockedBurnTime`

Severity: High Risk

Context: SonicAirdropNFT.sol#L28, SonicOrderBook.sol#L1431-L1445

Description: `safeTransferFrom` and `safeBatchTransferFrom` also transfer tokens after `lockedBurnTime` since they rely on `$._balances[id][from]` storage values in the `ERC1155Upgradeable` inherited contract and not the overwritten `balanceOf` of endpoint.

Proof of Concept: Apply the following test patch:

```
  diff --git a/test/sonicNFT/SonicAirdropNFT.ts b/test/sonicNFT/SonicAirdropNFT.ts
  index 9b37d03..b858a79 100644
-  -- a/test/sonicNFT/SonicAirdropNFT.ts
+  ++ b/test/sonicNFT/SonicAirdropNFT.ts
  @@ -612,6 +612,22 @@ describe("Sonic Airdrop NFT", () => {
         balance = await sonicAirdropNFT.balanceOf(user1, season);
         expect(balance).to.equal(0);
       });
+
+      it("should revert when transferring non-zero amount after burn time", async () => {
+        const {user1, user2, sonicAirdropNFT, season, seasonLockedBurnDate} = await loadFixture(fixture);
+
+        const origBalance = await sonicAirdropNFT.balanceOf(user1, season);
+        expect(origBalance).to.not.equal(0);
+
+        // Advance time past locked burn date
+        await ethers.provider.send("evm_setNextBlockTimestamp", [seasonLockedBurnDate + 1]);
+        await ethers.provider.send("evm_mine");
+
+        const balance = await sonicAirdropNFT.balanceOf(user1, season);
+        expect(balance).to.equal(0);
+
+        await sonicAirdropNFT.connect(user1).safeTransferFrom(user1, user2, season, origBalance, "0x");
+      });
     });

     describe("Burns", () => {
```

and run

```
yarn test -- --grep "should revert when transferring non-zero amount after burn time"
```

Recommendation: Make sure `safeTransferFrom` and `safeBatchTransferFrom` if a user tries to transfer a positive amount of a token id with `lockedBurnTime` in the past.

Paintswap: Fixed in PR 85.

Cantina Managed: Fix verified.

3.1.2 `placeMarketOrder()` sends incorrect amount of NFTs to users

Severity: High Risk

Context: SonicOrderBook.sol#L257-L259

Description: When the user buys market NFTs using `useExactQuantity` as false, the actual number of NFTs bought is `quantityFulfilled`, but the protocol sends `order.quantity` to the user, which may be smaller than `quantityFulfilled`.

Recommendation: Change to:

```
- uint256 nftAmountToTransfer = order.quantity - fee;
+ uint256 nftAmountToTransfer = quantityFulfilled - fee;
  _safeTransferNFTsFromUs(sender, order.tokenId, nftAmountToTransfer);
  _safeTransferNFTsFromUs(_devAddr, order.tokenId, fee);
```

Paintswap: Fixed in PR 87.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 `balanceOfBatch` does not consider the `lockedBurnTime` of the token ids when returning balances

Severity: Medium Risk

Context: SonicAirdropNFT.sol#L28

Description: `balanceOfBatch` does not consider the `lockedBurnTime` of the token ids when returning balances. So if the current timestamp is greater than a `lockedBurnTime` of the token id (`seasonId`), `balanceOfBatch` could return a positive value.

Recommendation: Make sure `balanceOfBatch` also returns `0` as balance of any user after the `lockedBurnTime` of the token id.

Paintswap: Fixed in PR 84.

Cantina Managed: Fix verified.

3.2.2 Disabling trading can be frontrun

Severity: Medium Risk

Context: SonicOrderBook.sol#L1549-L1556

Description: Assume the owner wants to set `tokenIdInfos[i].isTradeable` to `false` in transaction T_1 for example due to `lockedBurnTime` being in the past (although ideally one would want to disable trading before one approaches `lockedBurnTime` of a token id with some buffer). Then any user can frontrun T_1 with T_0 that would trade on the orderbook but these tokens would not have any values anymore.

Recommendation: To avoid the above issue. The owner would need to make sure to disable trading of token ids before they approach their `lockedBurnTime`. To be more secure one can also add a check in the for market and limit order flows to make sure no expired tokens are traded.

Related finding "`safeTransferFrom` and `safeBatchTransferFrom` also transfer tokens after `lockedBurnTime`", the fix for the finding should also solve this issue.

Footnote: The following endpoint was removed from the new scope which would also allow anyone within a certain buffer time to `lockedBurnTime` to disable trading those soon-to-be-expired tokens:

```
function refreshIsTradeable(uint256 tokenId) external nonReentrant {
  SeasonData memory seasonData = _nft.getSeasonData(Season(tokenId));
  // Check this season even exists
  require(seasonData.startTime != 0, SeasonDoesntExist(tokenId));

  if (seasonData.lockedBurnTime - SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME < block.timestamp) {
    // No longer tradeable 1 week before lock burn time
    _tokenIdInfos[tokenId].isTradeable = false;

    TokenIdInfo[] memory tokenIdInfos = new TokenIdInfo[](1);
    tokenIdInfos[0] = _tokenIdInfos[tokenId];

    uint256[] memory tokenIds = new uint256[](1);
    tokenIds[0] = tokenId;

    emit SetTokenIdInfos(tokenIds, tokenIdInfos);
  }
}
```

Paintswap: Acknowledged.

Cantina Managed: Acknowledged. The fix provided for "`safeTransferFrom` and `safeBatchTransferFrom` also transfer tokens after `lockedBurnTime`" would not completely solve the issue. The issue is kind of only fixed on one side for sell orders. But for buy orders, if filled it might not be able to be claimed.

3.2.3 The `useExactQuantity` feature is implemented incorrectly

Severity: Medium Risk

Context: SonicOrderBook.sol#L688, SonicOrderBook.sol#L747-L751, SonicOrderBook.sol#L834-L872

Description: When the order is of `Buy` type and `useExactQuantity` is false, the protocol want to spend all `order.totalCost` as much as possible, and `quantityFulfilled` is greater than or equal to `order.quantity`, `order.quantity` will be the minimum limit. But the actual implementation has the following problems.

1. `quantityFulfilled` cannot exceed `order.quantity`:

   When `maxAffordableQuantity` is greater than `quantityRemaining`, the protocol will only take `quantityRemaining`, not `maxAffordableQuantity`.

   ```
   if (maxAffordableQuantity >= quantityRemaining) {
     // We can afford the remaining quantity we want
     quantityNFTClaimable = quantityRemaining;
     quantityRemaining = 0;
   } else {
   ```

   And the protocol enforces that `quantityFulfilled` is less than or equal to `order.quantity`.

   ```
   quantityFulfilled = order.quantity - quantityRemaining;
   ```

2. When `quantityRemaining` is 0, the protocol will not continue to take orders to reach `maxCost`.

   Since the loop condition requires `quantityRemaining` to be not equal to 0, when `quantityRemaining` is equal to 0, even though `maxCost` still has space, the protocol won't take new orders.

   ```
   } else {
     // Eat into the order
     slot = slot.setQuantity(uint88(quantityL3 - quantityRemaining));
     segment = segment.setSlot(slotIndex, slot);
     quantityNFTClaimable = quantityRemaining;
     quantityRemaining = 0;
   }
   ```

3. DoS occurs when `maxAffordableQuantity` is less than `quantityRemaining`.

   When all budgets are used up, but `quantityRemaining` is not 0, the protocol will always access to the current price via `getLowestAsk()` in the loop, then causes DOS.

   ```
   if (maxAffordableQuantity == 0) {
     // Can't afford any of this order, stop here
     return (segment, quantityRemaining, ordersConsumed, quantityNFTClaimable, tradeCost, tradeFee);
   }

   // ...
   } else {
     // Take maximum we can afford
     quantityNFTClaimable = maxAffordableQuantity;
     quantityRemaining -= maxAffordableQuantity;
   }
   ```

Recommendation: The new `_parseOrder()` should implement as follows.

1. add `quantitySurplus` to represent the excess of `order.quantity`, then in `_placeMarketOrder()` adding `quantitySurplus` to `quantityFulfilled`.
2. when `maxAffordableQuantity` is less than `quantityRemaining`, since it can't satisfy `order.quantity`, just revert.

```
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -823,7 +823,8 @@ contract SonicOrderBook is
        uint256 ordersConsumed,
        uint256 quantityNFTClaimable,
        uint256 tradeCost,
-       uint256 tradeFee
+       uint256 tradeFee,
+       uint256 quantitySurplus
      )
    {
      bytes10 slot = segment.getSlot(slotIndex);
  @@ -837,18 +838,18 @@ contract SonicOrderBook is
        uint256 maxAffordableQuantity = (remainingBudget * BASE_DECIMAL_DIVISOR) / bestPrice;
        if (maxAffordableQuantity == 0) {
          // Can't afford any of this order, stop here
-         return (segment, quantityRemaining, ordersConsumed, quantityNFTClaimable, tradeCost, tradeFee);
+         revert("cant afford order.quantity"); // quantityRemaining != 0
        }

        // Take partial order based on budget
        if (maxAffordableQuantity >= quantityRemaining) {
          // We can afford the remaining quantity we want
-         quantityNFTClaimable = quantityRemaining;
+         quantityNFTClaimable = maxAffordableQuantity;
+         quantitySurplus = maxAffordableQuantity - quantityRemaining;
          quantityRemaining = 0;
        } else {
          // Take maximum we can afford
-         quantityNFTClaimable = maxAffordableQuantity;
-         quantityRemaining -= maxAffordableQuantity;
+         revert("cant afford order.quantity");
        }
```

And for the loop condition of `quantityRemaining != 0`, need to add a new condition of `maxAffordableQuantity != 0`.

```
```diff
-   while (quantityRemaining != 0) {
+   while (quantityRemaining != 0 || maxAffordableQuantity != 0) {
    // ...
-   for (uint256 i = node.tombstoneSegmentOffset; i < segments.length && quantityRemaining != 0; ++i) {
+   for (uint256 i = node.tombstoneSegmentOffset; i < segments.length && (quantityRemaining != 0 ||
↪  maxAffordableQuantity != 0) ; ++i) {
        lastConsumedSegmentIndex = i;
        segment = segments[i];

-       for (uint256 slotIndex = 0; slotIndex < NUM_SLOTS_PER_SEGMENT && quantityRemaining != 0; ++slotIndex) {
+       for (uint256 slotIndex = 0; slotIndex < NUM_SLOTS_PER_SEGMENT && (quantityRemaining != 0 ||
↪  maxAffordableQuantity != 0); ++slotIndex) {
```
```

Paintswap: Have made some updates in PR 87, but the changes made in the finding "The update of remaining quantity, cost and fee in the filling order flow make it hard to reason about invariants" superseed these.

Cantina Managed: Fix verified.

3.2.4 The update of remaining quantity, cost and fee in the filling order flow make it hard to reason about invariants

Severity: Medium Risk

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

Description: Currently, the update of remaining quantity, cost and fee in the filling order flow make it hard to reason about invariants. As it is demonstrated in "The `useExactQuantity` feature is implemented incorrectly" for example, certain new features are incorrectly implemented due to the above complexity.

Recommendation: To reduce the above complexity and also make it easier to prove the required invariants for the flows, it would be best to dedicate a specific region of memory for the important parameters in the order consumption flow:

```
struct TakeFromOrderBookParams {
  // the following fields should only be initialised once
  address sender;
  OrderSide side;
  uint256 tokenId;
  uint64 limitPrice;
  uint48 orderId;
  uint88 quantity;
  uint256 maxCost;
  bool useExactQuantity; // not really used right now
  OrderType orderType; // not really used right now
  bool isBudgetConstrainedBuying;

  // the following fields would change in memory
  uint64 bestPrice;
  uint256 quantityFulfilled;
  uint256 qunatityRemaining; // is not used, can be removed
  uint256 currentCost;
  uint256 fee;
  uint256 numberOfOrdersTraded;
  bool isBudgetConstrainedBuyingLimitReached;
}
```

Apply the following patch:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index 476c093..891b555 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -81,7 +81,7 @@ contract SonicOrderBook is
     error TooManyOrdersHit();
     error MaxOrdersNotMultipleOfOrdersInSegment();
     error ClaimingTooManyOrders();
-    error FailedToTakeFromBook(address taker, OrderSide side, uint256 tokenId, uint256 quantityRemaining);
+    error FailedToTakeExactQuantityFromBook(address taker, OrderSide side, uint256 tokenId, uint256
↪  quantityFulfilled, uint256 quantity);
     error TotalCostConditionNotMet();
     error TransferFailed();
     error NotTradeable();
  @@ -232,47 +232,57 @@ contract SonicOrderBook is
       uint48[] memory orderIdsPool = new uint48[](MAX_ORDERS_HIT);
       uint256[] memory quantitiesPool = new uint256[](MAX_ORDERS_HIT);

-      (uint256 cost, uint256 fee, uint256 quantityFulfilled) = _placeMarketOrder(
+      (uint256 cost, uint256 fee, uint256 quantityFulfilled, bool isBudgetConstrainedBuying) =
↪  _placeMarketOrder(
         sender,
         order,
         orderIdsPool,
         quantitiesPool
       );
-      bool isBuy = order.side == OrderSide.Buy;
-      uint256 quoteFee;
-      if (isBuy) {
-        if (order.useExactQuantity) {
-          require(cost <= order.totalCost, TotalCostConditionNotMet());
-          quoteTokensFromUs = 0;
-          quoteTokensToUs = cost;
-        } else {
-          // Budget-constrained buying: ensure minimum quantity was met
+
+      if (order.side == OrderSide.Buy) {
+        // Potentially the below check could also be conditioned based on `isBudgetConstrainedBuying`
+        // It all depends on how one wants the `order.totalCost` to limit our order execution
+        // The check below should already hold when `isBudgetConstrainedBuying` is `true`
+        require(cost <= order.totalCost, TotalCostConditionNotMet());
+
+        /*  if the order is a non-exact buy marker order, aka constraint-budget buy order
+         *  then it would be possible that `quantityFulfilled` is less than or equal or greater than
+         *  `order.quantity`. If desired we can remove the `if` statement below and instead check
+         *  the condition always no matter what value `isBudgetConstrainedBuying` would have.
+         *  Then`isBudgetConstrainedBuying` return value can be removed from `_placeMarketOrder`'s
+         *  return parameters. Note that `isBudgetConstrainedBuying` still needs to be used internally in
+         *  `_placeMarketOrder`'s flow when one is taking from the orderbook.
+         */
```

```
+
+        if(!isBudgetConstrainedBuying) {
+          // we are allowing for non-exact buy marker orders, aka constraint-budget buy orders,
+          // the `quantityFulfilled` to potentially fall below `order.quantity`
+          // that is why the check below is within the `if` block.
+          // below should be guaranteed in this case since the order would need to be exact
+          // but conditions might change in the future
           require(quantityFulfilled >= order.quantity, InsufficientQuantityFulfilled());
-          // Refund the difference between budget and actual cost
-          quoteTokensFromUs = order.totalCost > cost ? order.totalCost - cost : 0;
-          quoteTokensToUs = order.totalCost; // Take the full budget amount
         }

+        quoteTokensFromUs = 0;
+        quoteTokensToUs = cost;
+
         // Transfer NFTs to buyer (minus NFT fees if buying)
-        uint256 nftAmountToTransfer = order.quantity - fee;
-        _safeTransferNFTsFromUs(sender, order.tokenId, nftAmountToTransfer);
+        uint256 nftAmountToTransfer = quantityFulfilled - fee;
+
+        _resolveQuoteTokens(sender, 0, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
         _safeTransferNFTsFromUs(_devAddr, order.tokenId, fee);
+        _safeTransferNFTsFromUs(sender, order.tokenId, nftAmountToTransfer);
       } else {
-        if (order.useExactQuantity) {
-          require(cost >= order.totalCost, TotalCostConditionNotMet());
-        } else {
-          // Ensure minimum funds were met
-          require(cost - fee >= order.totalCost, InsufficientQuoteReceived());
-        }
+        require(cost - fee >= order.totalCost, InsufficientQuoteReceived());
+
         // Transfer tokens to the seller if any have sold
         quoteTokensFromUs = cost - fee;
         quoteTokensToUs = 0;
-        quoteFee = fee;
+
         // Buying transfer the NFTs to the taker
         _safeTransferNFTsToUs(sender, order.tokenId, order.quantity);
+        _resolveQuoteTokens(sender, fee, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
       }
-
-      _resolveQuoteTokens(sender, quoteFee, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
     }

     /// @notice Place multiple limit orders in the order book
  @@ -565,6 +575,17 @@ contract SonicOrderBook is
       // Assumes up to 5 seasons
       uint256[5] memory nftFeeInSeasons;

+      // Allocate a memory region to be used when taking from orderbook.
+      // This memory region will be reused in each iteration, preventing
+      // unnecessary memory expansions.
+      // Note make sure to reset and repopulate the other fields in each iteration
+      TakeFromOrderBookParams memory params;
+      params.sender = sender;
+      params.maxCost = type(uint256).max;
+      params.useExactQuantity = false;
+      params.orderType = OrderType.LIMIT_ORDER;
+      params.isBudgetConstrainedBuying = false;
+
       for (uint256 i = 0; i < orders.length; ++i) {
         LimitOrder calldata order = orders[i];
         uint128 priceTick = _tokenIdInfos[order.tokenId].priceTick;
  @@ -574,7 +595,7 @@ contract SonicOrderBook is
           uint256 cost,
           uint64 price,
           uint256 takerFee
-        ) = _placeLimitOrder(sender, currentOrderId, order, orderIdsPool, quantitiesPool, priceTick);
+        ) = _placeLimitOrder(params, currentOrderId, order, orderIdsPool, quantitiesPool, priceTick);
         ++currentOrderId;

         if (order.side == OrderSide.Buy) {
  @@ -662,94 +683,117 @@ contract SonicOrderBook is
       emit OrdersCancelled(sender, orderIds);
```

```
    }

+   function _shouldTakeMoreFromOrderBook(
+     TakeFromOrderBookParams memory params
+   ) internal pure returns (bool) {
+     if (params.isBudgetConstrainedBuyingLimitReached) {
+       return false;
+     }
+
+     if (params.isBudgetConstrainedBuying) {
+       return params.currentCost < params.maxCost;
+     } else {
+       return params.quantityFulfilled < params.quantity;
+     }
+   }
+
+   function _tryUpdateBestPriceAndContinue(
+     TakeFromOrderBookParams memory params
+   ) internal view returns (bool) {
+     bool senderIsSelling = params.side == OrderSide.Sell;
+     uint64 bestPrice = senderIsSelling ?
+       getHighestBid(params.tokenId) : getLowestAsk(params.tokenId);
+
+     if (bestPrice == 0) {
+       return false;
+     }
+
+     if (senderIsSelling) {
+       if (bestPrice < params.limitPrice) {
+         return false;
+       }
+     } else {
+       // sender is buying
+       if (bestPrice > params.limitPrice) {
+         return false;
+       }
+     }
+
+     // at this point we know that
+     // 1. bestPrice != 0
+     // 2. bestPrice satisfy the bound set by the limitPrice
+
+     params.bestPrice = bestPrice;
+     return true;
+   }
+
    function _takeFromOrderBookSide(
-     address sender,
-     uint256 tokenId,
-     uint64 price,
-     uint256 quantity,
+     TakeFromOrderBookParams memory params,
      uint48[] memory orderIdsPool,
      uint256[] memory quantitiesPool,
-     OrderSide side, // which side are you taking from
-     uint48 takerOrderId,
      mapping(uint256 tokenId => mapping(uint256 price => bytes32[] segments)) storage segmentsAtPrice,
-     mapping(uint256 tokenId => BokkyPooBahsRedBlackTreeLibrary.Tree) storage tree,
-     uint256 maxCost
-   ) private returns (uint256 quantityRemaining, uint256 cost, uint256 fee) {
-     quantityRemaining = quantity;
-
+     mapping(uint256 tokenId => BokkyPooBahsRedBlackTreeLibrary.Tree) storage tree
+   ) private {
      // reset the size
      assembly ("memory-safe") {
        mstore(orderIdsPool, MAX_ORDERS_HIT)
        mstore(quantitiesPool, MAX_ORDERS_HIT)
      }

-     bool isTakingFromBuy = side == OrderSide.Buy;
-     uint256 numberOfOrders = 0;
-     while (quantityRemaining != 0) {
-       uint64 bestPrice = isTakingFromBuy ? getHighestBid(tokenId) : getLowestAsk(tokenId);
-       if (bestPrice == 0 || (isTakingFromBuy ? bestPrice < price : bestPrice > price)) {
-         // No more orders left
```

```
-         break;
-       }
-
-       uint256 currentCost = 0;
-       uint256 currentFee = 0;
-       (quantityRemaining, currentCost, currentFee, numberOfOrders) = _consumePriceSegments(
-         tree[tokenId],
-         segmentsAtPrice[tokenId][bestPrice],
-         bestPrice,
-         quantityRemaining,
+     while (
+       _shouldTakeMoreFromOrderBook(params) &&
+       _tryUpdateBestPriceAndContinue(params)
+     ) {
+       _consumePriceSegments(
+         params,
+         tree[params.tokenId],
+         segmentsAtPrice[params.tokenId][params.bestPrice],
          orderIdsPool,
-         quantitiesPool,
-         numberOfOrders,
-         maxCost,
-         cost,
-         isTakingFromBuy
+         quantitiesPool
        );
-
-       cost += currentCost;
-       fee += currentFee;
-
-       // Check if we've hit the budget limit
-       if (cost >= maxCost) {
-         break;
-       }
      }

-     if (numberOfOrders != 0) {
+     uint256 numberOfOrdersTraded = params.numberOfOrdersTraded;
+     if (numberOfOrdersTraded != 0) {
        assembly ("memory-safe") {
-         mstore(orderIdsPool, numberOfOrders)
-         mstore(quantitiesPool, numberOfOrders)
+         mstore(orderIdsPool, numberOfOrdersTraded)
+         mstore(quantitiesPool, numberOfOrdersTraded)
        }

-       emit OrdersMatched(sender, takerOrderId, orderIdsPool, quantitiesPool);
+       emit OrdersMatched(params.sender, params.orderId, orderIdsPool, quantitiesPool);
      }
    }

    function _consumePriceSegments(
+     TakeFromOrderBookParams memory params,
      BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
      bytes32[] storage segments,
-     uint64 bestPrice,
-     uint256 quantityRemaining,
      uint48[] memory orderIdsPool,
-     uint256[] memory quantitiesPool,
-     uint256 numberOfOrders,
-     uint256 maxCost,
-     uint256 currentTotalCost,
-     bool isTakingFromBuy
-   ) private returns (uint256 newQuantityRemaining, uint256 cost, uint256 fee, uint256 newNumberOfOrders) {
-     BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree.getNodeUnsafe(bestPrice);
+     uint256[] memory quantitiesPool
+   ) private {
+     // when bestPrice (!=0) was picked it should be guaranteed that the node exists.
+     BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree.getNodeUnsafe(params.bestPrice);

      uint256 numOrdersFullyConsumed = 0;
      bytes32 segment;
      uint256 lastConsumedSegmentIndex;

-     for (uint256 i = node.tombstoneSegmentOffset; i < segments.length && quantityRemaining != 0; ++i) {
+     for (
```

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

-        for (uint256 slotIndex = 0; slotIndex < NUM_SLOTS_PER_SEGMENT && quantityRemaining != 0; ++slotIndex) {
-          require(numberOfOrders < MAX_ORDERS_HIT, TooManyOrdersHit());
+        for (
+          uint256 slotIndex = 0;
+          slotIndex < NUM_SLOTS_PER_SEGMENT &&
+            _shouldTakeMoreFromOrderBook(params);
+          ++slotIndex
+        ) {
+          require(params.numberOfOrdersTraded < MAX_ORDERS_HIT, TooManyOrdersHit());
           bytes32 remainingSegment = segment.getRemainingSegment(slotIndex);
           if (remainingSegment == 0) {
             // Should only hit this branch on the last segment
  @@ -766,101 +810,89 @@ contract SonicOrderBook is

           uint256 ordersConsumed = 0;
           uint256 quantityNFTClaimable = 0;
-          uint256 tradeCost = 0;
-          uint256 tradeFee = 0;
-          (segment, quantityRemaining, ordersConsumed, quantityNFTClaimable, tradeCost, tradeFee) =
↪  _parseOrder(
-            bestPrice,
+          (segment, ordersConsumed, quantityNFTClaimable) = _parseOrder(
+            params,
             segment,
             slotIndex,
-            quantityRemaining,
-            maxCost,
-            currentTotalCost + cost,
-            _devFeeTakerBps,
-            isTakingFromBuy
+            _devFeeTakerBps
           );

-          cost += tradeCost;
-          fee += tradeFee;
           numOrdersFullyConsumed += ordersConsumed;

-          orderIdsPool[numberOfOrders] = orderId;
-          quantitiesPool[numberOfOrders] = quantityNFTClaimable;
-          ++numberOfOrders;
-
-          // Check if we've hit the budget limit
-          if (currentTotalCost + cost >= maxCost) {
-            break;
-          }
-        }
-
-        // Check if we've hit the budget limit
-        if (currentTotalCost + cost >= maxCost) {
-          break;
+          orderIdsPool[params.numberOfOrdersTraded] = orderId;
+          quantitiesPool[params.numberOfOrdersTraded] = quantityNFTClaimable;
+          params.numberOfOrdersTraded += 1;
         }
       }

-      _updateSegmentsAndTree(tree, node, segments, bestPrice, segment, lastConsumedSegmentIndex,
↪  numOrdersFullyConsumed);
-
-      newQuantityRemaining = quantityRemaining;
-      newNumberOfOrders = numberOfOrders;
+      _updateSegmentsAndTree(
+        tree,
+        node,
+        segments,
+        params.bestPrice,
+        segment,
+        lastConsumedSegmentIndex,
```

```
+         numOrdersFullyConsumed
+       );
      }

      function _parseOrder(
-       uint256 bestPrice,
+       TakeFromOrderBookParams memory params,
        bytes32 segment,
        uint256 slotIndex,
-       uint256 quantityRemaining,
-       uint256 maxCost,
-       uint256 currentTotalCost,
-       uint256 devFeeTakerBps,
-       bool isTakingFromBuy
+       uint256 devFeeTakerBps
      )
        private
        view
        returns (
          bytes32 newSegment,
-         uint256 newQuantityRemaining,
          uint256 ordersConsumed,
-         uint256 quantityNFTClaimable,
-         uint256 tradeCost,
-         uint256 tradeFee
+         uint256 quantityNFTClaimable
        )
      {
        bytes10 slot = segment.getSlot(slotIndex);
        uint256 quantityL3 = slot.getQuantity();

-       // Check budget constraint before consuming order
-       uint256 fullOrderCost = (quantityL3 * bestPrice) / BASE_DECIMAL_DIVISOR;
-       if (currentTotalCost + fullOrderCost > maxCost) {
-         // Check if we can afford partial order
-         uint256 remainingBudget = maxCost - currentTotalCost;
-         uint256 maxAffordableQuantity = (remainingBudget * BASE_DECIMAL_DIVISOR) / bestPrice;
-         if (maxAffordableQuantity == 0) {
-           // Can't afford any of this order, stop here
-           return (segment, quantityRemaining, ordersConsumed, quantityNFTClaimable, tradeCost, tradeFee);
-         }
+       if (params.isBudgetConstrainedBuying) {
+         // Check budget constraint before consuming order
+         uint256 fullOrderCost = (quantityL3 * params.bestPrice) / BASE_DECIMAL_DIVISOR;
+         uint256 remainingBudget = params.maxCost - params.currentCost;
+
+         if (fullOrderCost > remainingBudget) {
+           // Check if we can afford partial order
+           uint256 maxAffordableQuantity = (remainingBudget * BASE_DECIMAL_DIVISOR) / params.bestPrice;
+           if (maxAffordableQuantity == 0) {
+             // Can't afford any of this order, stop here
+             params.isBudgetConstrainedBuyingLimitReached = true;
+             return (
+               segment,
+               0, // ordersConsumed
+               0 // quantityNFTClaimable
+             );
+           }

-         // Take partial order based on budget
-         if (maxAffordableQuantity >= quantityRemaining) {
-           // We can afford the remaining quantity we want
-           quantityNFTClaimable = quantityRemaining;
-           quantityRemaining = 0;
-         } else {
-           // Take maximum we can afford
            quantityNFTClaimable = maxAffordableQuantity;
-           quantityRemaining -= maxAffordableQuantity;
+
+           // Update the slot with remaining quantity
+           slot = slot.setQuantity(uint88(quantityL3 - quantityNFTClaimable));
+           segment = segment.setSlot(slotIndex, slot);
+         } else {
+           // the remaining budget can afford consuming the whole order
+           quantityNFTClaimable = quantityL3;
+           ordersConsumed = 1;
```

```
        }

-       // Update the slot with remaining quantity
-       slot = slot.setQuantity(uint88(quantityL3 - quantityNFTClaimable));
-       segment = segment.setSlot(slotIndex, slot);
+       // at this point we shold have
+       // params.currentCost + (quantityNFTClaimable * params.bestPrice) / BASE_DECIMAL_DIVISOR
+       // <= params.maxCost
+       //
+       // So in general during the whole iteration of matching orders we should have:
+       // params.currentCost <= params.maxCost
      } else {
-       // No budget constraint or we can afford this order
+       // non-budget constrained order. Need to make sure we don't
+       // fill/match more than params.quantity
+       uint256 quantityRemaining = params.quantity - params.quantityFulfilled;
        if (quantityRemaining >= quantityL3) {
-         // Consume this whole order
-         unchecked {
-           quantityRemaining -= quantityL3;
-         }
          quantityNFTClaimable = quantityL3;
          ordersConsumed = 1;
        } else {
@@ -868,21 +900,29 @@ contract SonicOrderBook is
          slot = slot.setQuantity(uint88(quantityL3 - quantityRemaining));
          segment = segment.setSlot(slotIndex, slot);
          quantityNFTClaimable = quantityRemaining;
-         quantityRemaining = 0;
        }
+
+       // at this point one should be able to prove that:
+       // params.quantityFulfilled + quantityNFTClaimable <= params.quantity
+       // so for non-budget constrained orders we should always have:
+       // params.quantityFulfilled <= params.quantity
+       //
+       // Only non-exact buy market orders can be budget constrained.
      }

-     tradeCost = (quantityNFTClaimable * bestPrice) / BASE_DECIMAL_DIVISOR;
+     { // update quantityFulfilled, currentCost and fee
+       params.quantityFulfilled += quantityNFTClaimable;
+       uint256 tradeCost = (quantityNFTClaimable * params.bestPrice) / BASE_DECIMAL_DIVISOR;
+       params.currentCost += tradeCost;

-     if (isTakingFromBuy) {
-       tradeFee = (tradeCost * devFeeTakerBps) / FEE_SCALING_FACTOR;
-     } else {
-       // If the order is a sell order, the maker pays the dev fee
-       tradeFee = (quantityNFTClaimable * devFeeTakerBps) / FEE_SCALING_FACTOR;
+       if (params.side == OrderSide.Sell) {
+         params.fee += (tradeCost * devFeeTakerBps) / FEE_SCALING_FACTOR;
+       } else {
+         params.fee += (quantityNFTClaimable * devFeeTakerBps) / FEE_SCALING_FACTOR;
+       }
      }

      newSegment = segment;
-     newQuantityRemaining = quantityRemaining;
    }

    function _updateSegmentsAndTree(
@@ -931,44 +971,26 @@ contract SonicOrderBook is
    }

    function _takeFromOrderBook(
-     address sender,
-     OrderSide side,
-     uint256 tokenId,
-     uint64 price,
-     uint88 quantity,
-     uint48 takerOrderId,
+     TakeFromOrderBookParams memory params,
      uint48[] memory orderIdsPool,
-     uint256[] memory quantitiesPool,
-     uint256 maxCost
```

```
-    ) private returns (uint256 quantityRemaining, uint256 cost, uint256 fee) {
+      uint256[] memory quantitiesPool
+    ) private {
       // Take as much as possible from the order book
-      if (side == OrderSide.Buy) {
-        (quantityRemaining, cost, fee) = _takeFromOrderBookSide(
-          sender,
-          tokenId,
-          price,
-          quantity,
+      if (params.side == OrderSide.Buy) {
+        _takeFromOrderBookSide(
+          params,
           orderIdsPool,
           quantitiesPool,
-          OrderSide.Sell,
-          takerOrderId,
           _asksAtPrice,
-          _asks,
-          maxCost
+          _asks
         );
       } else {
-        (quantityRemaining, cost, fee) = _takeFromOrderBookSide(
-          sender,
-          tokenId,
-          price,
-          quantity,
+        _takeFromOrderBookSide(
+          params,
           orderIdsPool,
           quantitiesPool,
-          OrderSide.Buy,
-          takerOrderId,
           _bidsAtPrice,
-          _bids,
-          type(uint256).max // No budget constraint for selling
+          _bids
         );
       }
     }
  @@ -1023,12 +1045,40 @@ contract SonicOrderBook is
       _tokenClaimables[uint48(orderId)].remainingNormalizedQuantity -= uint32(quantity / QUANTITY_TICK);
     }

+    enum OrderType {
+      MARKET_ORDER,
+      LIMIT_ORDER
+    }
+
+    struct TakeFromOrderBookParams {
+      // the following fields should only be initialised once
+      address sender;
+      OrderSide side;
+      uint256 tokenId;
+      uint64 limitPrice;
+      uint48 orderId;
+      uint88 quantity;
+      uint256 maxCost;
+      bool useExactQuantity; // not really used right now
+      OrderType orderType; // not really used right now
+      bool isBudgetConstrainedBuying;
+
+      // the following fields would change in memory
+      uint64 bestPrice;
+      uint256 quantityFulfilled;
+      uint256 qunatityRemaining; // is not used, can be removed
+      uint256 currentCost;
+      uint256 fee;
+      uint256 numberOfOrdersTraded;
+      bool isBudgetConstrainedBuyingLimitReached;
+    }
+
     function _placeMarketOrder(
       address sender,
       MarketOrder calldata order,
```

```
      uint48[] memory orderIdsPool,
      uint256[] memory quantitiesPool
-    ) private returns (uint256 cost, uint256 fee, uint256 quantityFulfilled) {
+    ) private returns (uint256 cost, uint256 fee, uint256 quantityFulfilled, bool isBudgetConstrainedBuying) {
      require(order.quantity != 0, NoQuantity());

      TokenIdInfo memory tokenIdInfo = _tokenIdInfos[order.tokenId];
  @@ -1038,37 +1088,53 @@ contract SonicOrderBook is
      uint48 newOrderId = _nextOrderId++;
      emit NewOrder(sender, newOrderId);

-     uint256 quantityRemaining;
      uint64 price = order.side == OrderSide.Buy ? type(uint64).max : 0;

      uint256 maxCost = type(uint256).max;
      if (!order.useExactQuantity && order.side == OrderSide.Buy) {
        // Budget-constrained buying: use totalCost as maximum budget
        maxCost = order.totalCost;
+       isBudgetConstrainedBuying = true;
      }

-     (quantityRemaining, cost, fee) = _takeFromOrderBook(
-       sender,
-       order.side,
-       order.tokenId,
-       price,
-       order.quantity,
-       newOrderId,
-       orderIdsPool,
-       quantitiesPool,
-       maxCost
-     );
+     TakeFromOrderBookParams memory params = TakeFromOrderBookParams({
+       sender: sender,
+       side: order.side,
+       tokenId: order.tokenId,
+       limitPrice: price, // should make the price limit check trivial since extreme endpoints are chosen
↪  based on the order side
+       orderId: newOrderId,
+       quantity: order.quantity,
+       maxCost: maxCost,
+       useExactQuantity: order.useExactQuantity,
+       orderType: OrderType.MARKET_ORDER,
+       isBudgetConstrainedBuying: isBudgetConstrainedBuying,
+       // the following would change in memory
+       bestPrice: 0,
+       quantityFulfilled: 0,
+       qunatityRemaining: order.quantity,
+       currentCost: 0,
+       fee: 0,
+       numberOfOrdersTraded: 0,
+       isBudgetConstrainedBuyingLimitReached: false
+     });

-     quantityFulfilled = order.quantity - quantityRemaining;
+     _takeFromOrderBook(params, orderIdsPool, quantitiesPool);
+
+     cost = params.currentCost;
+     fee = params.fee;
+     quantityFulfilled = params.quantityFulfilled;

      if (order.useExactQuantity) {
        // Must fulfill complete order
-       require(quantityRemaining == 0, FailedToTakeFromBook(sender, order.side, order.tokenId,
↪  quantityRemaining));
+       require(
+         quantityFulfilled == order.quantity,
+         FailedToTakeExactQuantityFromBook(sender, order.side, order.tokenId, quantityFulfilled,
↪  order.quantity)
+       );
      }
    }

    function _placeLimitOrder(
-     address sender,
+     TakeFromOrderBookParams memory params,
```

```
      uint48 newOrderId,
      LimitOrder calldata order,
      uint48[] memory orderIdsPool,
  @@ -1082,33 +1148,44 @@ contract SonicOrderBook is
      require(_tokenIdInfos[order.tokenId].isTradeable, NotTradeable());
      require(order.price % priceTick == 0, PriceNotMultipleOfTick(priceTick));

-      emit NewOrder(sender, newOrderId);
+      emit NewOrder(params.sender, newOrderId);
+
+      {// the following fields need to be populated or rest
+        params.side = order.side;
+        params.tokenId = order.tokenId;
+        params.limitPrice = order.price;
+        params.orderId = newOrderId;
+        params.quantity = order.quantity;
+
+        // the following would change in memory
+        params.bestPrice = 0;
+        params.quantityFulfilled = 0;
+        params.qunatityRemaining = order.quantity;
+        params.currentCost = 0;
+        params.fee = 0;
+        params.numberOfOrdersTraded = 0;
+        params.isBudgetConstrainedBuyingLimitReached = false;
+      }

-      uint256 quantityRemaining;
-      (quantityRemaining, cost, fee) = _takeFromOrderBook(
-        sender,
-        order.side,
-        order.tokenId,
-        order.price,
-        order.quantity,
-        newOrderId,
-        orderIdsPool,
-        quantitiesPool,
-        type(uint256).max // No budget constraint for limit orders
-      );
+      _takeFromOrderBook(params, orderIdsPool, quantitiesPool);
+
+      cost = params.currentCost;
+      fee = params.fee;
+      uint256 quantityFulfilled = params.quantityFulfilled;

        if (cost != 0) {
          require(!order.postOnly, PostOnly());
        }

+      uint256 quantityRemaining = order.quantity - quantityFulfilled;
        // Add the rest to the order book if has the minimum required, in order to keep order books healthy
        if (quantityRemaining >= _tokenIdInfos[order.tokenId].minQuantity) {
          quantityAddedToBook = quantityRemaining;
-         price = _addToBook(sender, newOrderId, priceTick, order.side, order.tokenId, order.price,
↪  quantityAddedToBook);
+         price = _addToBook(params.sender, newOrderId, priceTick, order.side, order.tokenId, order.price,
↪  quantityAddedToBook);
        } else if (quantityRemaining != 0) {
          price = order.price;
          failedQuantity = quantityRemaining;
-         emit FailedToAddToBook(sender, order.side, newOrderId, order.tokenId, price, failedQuantity);
+         emit FailedToAddToBook(params.sender, order.side, newOrderId, order.tokenId, price, failedQuantity);
        }
      }

  diff --git a/test/sonicOrderBook/SonicOrderBook.ts b/test/sonicOrderBook/SonicOrderBook.ts
  index 23b2b1c..a268c4d 100644
-  -- a/test/sonicOrderBook/SonicOrderBook.ts
+  ++ b/test/sonicOrderBook/SonicOrderBook.ts
  @@ -434,7 +434,7 @@ describe("SonicOrderBook", function () {
              },
              RETURN_WRAPPED_SONIC
            )
-         ).to.be.revertedWithCustomError(orderBook, "FailedToTakeFromBook");
+         ).to.be.revertedWithCustomError(orderBook, "FailedToTakeExactQuantityFromBook");
        });
```

```
      it("Take from buy order book, 1 exact order among a segment", async function () {
  @@ -561,7 +561,7 @@ describe("SonicOrderBook", function () {
              },
              RETURN_WRAPPED_SONIC
            )
-         ).to.be.revertedWithCustomError(orderBook, "FailedToTakeFromBook");
+         ).to.be.revertedWithCustomError(orderBook, "FailedToTakeExactQuantityFromBook");
       });

       it("Take from buy order book, max orders hit should revert", async function () {
```

Footnote:

- Some comments has been added to potentially alter/customise the route for budget-constrained buy market orders.
- Recommendation from "`placeMarketOrder()` sends incorrect amount of NFTs to users" has been applied in this patch.
- Modified version of the recommendation from "The distribution branches after placing the market order can be simplified" has been applied in this patch.

Paintswap: We had to apply the patch changes manually due to some conflict, so please double check it looks correct. We removed the unused variables which had comments next to them from the new taker param.

We think the condition in `placeMarketOrder` should be inverted? `if (!isBudgetConstrainedBuying)` to `if (isBudgetConstrainedBuying)`.

Also to keep quantity aligned to the quantity tick we have added this in `_parseOrder`:

```
// Floor to QUANTITY_TICK to ensure proper alignment
maxAffordableQuantity = (maxAffordableQuantity / QUANTITY_TICK) * QUANTITY_TICK;
```

We have used test updates from the finding "The useExactQuantity feature is implemented incorrectly" to prove the `useExactQuantity`.

Updates in PR 89

Cantina Managed: Fix verified. Note: When selling NFTs, the quote token received may be less than the totalCost (slippage control does not take fee into account).

3.3 Low Risk

3.3.1 `setBanned` can be frontrun

Severity: Low Risk

Context: SonicAirdropNFT.sol#L550-L559

Description: If a to-be-banned account can monitor the mempool and sees its account is about to be banned through the `setBanned` endpoint, it can transfer its tokens to another account.

Recommendation: To avoid the above issue one solution would be to submit the `setBanned` calls in a private way so that the accounts would not be able to detect them before it becoming part of a finalised block.

Paintswap: Acknowledged. While it may be possible theoretically on Sonic, I don't think it is very practical due to how the blocks are formed. They aren't done like blocks on Ethereum for instance where a leader is assigned a slot to put different transactions from the mempool in to form the block. On Sonic transactions get added to a dag first in a multi-leader way and then a "fake" block gets created around various transactions using some math so that all nodes create the same blocks with the same transaction ordering.

Worst case, pausing transfers can be done first, evaluate if the address is the same that needs banning (in case they transferred away right before), ban them and then unpause transfers.

Cantina Managed: Acknowledged.

3.3.2 `isApprovedForAll` does not check for banned operators

Severity: Low Risk

Context: SonicAirdropNFT.sol#L430-L439

Description: `isApprovedForAll` does not check for banned operators. So a banned operator account can transfer tokens for a non-banned owner account if approval is given to the operator.

Recommendation: For compliance reasons one might also would want to restrict access to banned operators. Moreover it might be necessary to also disallow calling `setApprovalForAll` for a banned operator or owner.

Paintswap: Fixed in PR 83.

Cantina Managed: Fix verified.

3.3.3 The distribution branches after placing the market order can be simplified

Severity: Low Risk

Context: SonicOrderBook.sol#L243-L275

Description:

1. When a non-exact buy market order is placed there is the following line which makes one think that in general `cost` could be greater than `order.totalCost` but this is not the case:

```
quoteTokensFromUs = order.totalCost > cost ? order.totalCost - cost : 0;
quoteTokensToUs = order.totalCost; // Take the full budget amount
```

2. For the sell market orders the lower bound check for exact versus non-exact orders are different. The bound check for non-exact sell market orders are more strict.
3. For buy market orders NFT tokens are sent to the user first then the quote tokens are acquired from the user. It would be best to always acquire from the user first then send the corresponding assets to the user.

Recommendation: The suggestion is to change the `if/else` branch to:

```
if (isBuy) {
  require(cost <= order.totalCost, TotalCostConditionNotMet());
  require(quantityFulfilled >= order.quantity, InsufficientQuantityFulfilled());

  quoteTokensFromUs = 0;
  quoteTokensToUs = cost;

  // Transfer NFTs to buyer (minus NFT fees if buying)
  uint256 nftAmountToTransfer = quantityFulfilled - fee;

  _resolveQuoteTokens(sender, quoteFee, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
  _safeTransferNFTsFromUs(_devAddr, order.tokenId, fee);
  _safeTransferNFTsFromUs(sender, order.tokenId, nftAmountToTransfer);
} else {
  require(cost - fee >= order.totalCost, InsufficientQuoteReceived());

  // Transfer tokens to the seller if any have sold
  quoteTokensFromUs = cost - fee;
  quoteTokensToUs = 0;
  quoteFee = fee;
  // Buying transfer the NFTs to the taker
  _safeTransferNFTsToUs(sender, order.tokenId, order.quantity);
  _resolveQuoteTokens(sender, quoteFee, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
}
```

Note that how we have removed the inner `if/else` blocks to unify the flows for exact and non-exact orders. This means that the following checks are always performed for any buy market order:

```
require(cost <= order.totalCost, TotalCostConditionNotMet());
require(quantityFulfilled >= order.quantity, InsufficientQuantityFulfilled());
```

The stricter check is used for the sell market orders:

```
require(cost - fee >= order.totalCost, InsufficientQuoteReceived());
```

and handling of assets is rearranged so that the orderbook receives first then the user. The suggestion from the "placeMarketOrder() sends incorrect amount of NFTs to users" is also applied. Apply the following patch:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index 476c093..832438b 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -241,38 +241,29 @@ contract SonicOrderBook is
       bool isBuy = order.side == OrderSide.Buy;
       uint256 quoteFee;
       if (isBuy) {
-         if (order.useExactQuantity) {
-           require(cost <= order.totalCost, TotalCostConditionNotMet());
-           quoteTokensFromUs = 0;
-           quoteTokensToUs = cost;
-         } else {
-           // Budget-constrained buying: ensure minimum quantity was met
-           require(quantityFulfilled >= order.quantity, InsufficientQuantityFulfilled());
-           // Refund the difference between budget and actual cost
-           quoteTokensFromUs = order.totalCost > cost ? order.totalCost - cost : 0;
-           quoteTokensToUs = order.totalCost; // Take the full budget amount
-         }
+         require(cost <= order.totalCost, TotalCostConditionNotMet());
+         require(quantityFulfilled >= order.quantity, InsufficientQuantityFulfilled());
+
+         quoteTokensFromUs = 0;
+         quoteTokensToUs = cost;

          // Transfer NFTs to buyer (minus NFT fees if buying)
-         uint256 nftAmountToTransfer = order.quantity - fee;
-         _safeTransferNFTsFromUs(sender, order.tokenId, nftAmountToTransfer);
+         uint256 nftAmountToTransfer = quantityFulfilled - fee;
+
+         _resolveQuoteTokens(sender, quoteFee, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
          _safeTransferNFTsFromUs(_devAddr, order.tokenId, fee);
+         _safeTransferNFTsFromUs(sender, order.tokenId, nftAmountToTransfer);
       } else {
-         if (order.useExactQuantity) {
-           require(cost >= order.totalCost, TotalCostConditionNotMet());
-         } else {
-           // Ensure minimum funds were met
-           require(cost - fee >= order.totalCost, InsufficientQuoteReceived());
-         }
+         require(cost - fee >= order.totalCost, InsufficientQuoteReceived());
+
          // Transfer tokens to the seller if any have sold
          quoteTokensFromUs = cost - fee;
          quoteTokensToUs = 0;
          quoteFee = fee;
          // Buying transfer the NFTs to the taker
          _safeTransferNFTsToUs(sender, order.tokenId, order.quantity);
+         _resolveQuoteTokens(sender, quoteFee, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
       }
-
-       _resolveQuoteTokens(sender, quoteFee, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
     }

     /// @notice Place multiple limit orders in the order book
```

Paintswap: Fixed in PR 88.

Cantina Managed: Fix verified. Note: When selling NFTs, the quote token received may be less than the totalCost (slippage control does not take fee into account).

3.3.4 Interactions of the banned users with the orderbook

Severity: Low Risk

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

Description: The banning functionality of the `_nft` and the orderbook play as follows:

1. If a user gets banned and tries to place a sell or buy market order, the transaction would fail due to the final transfer of the `_nft` from or to the user failing. This is not an issue.
2. If a user gets banned and tries to place a sell limit order, the transaction would fail since the final transfer of NFT tokens from the user to the orderbook would fail. This is not an issue.
3. If a user gets banned and tries to place a buy limit order, the transaction would go through (if not immediately partially matched/filled), but later when it gets matched and the user tries to claim its nft tokens the transfer would fail.
4. If a user places a sell limit order and later the user gets banned. When this order gets matched/filled (the NFT tokens are already transferred to the orderbook so the transfer to the buyer/taker goes through), the user/seller can claim its quote tokens.
5. If a user places a buy limit order and later the user gets banned. When this order gets matched/filled, the user would not be able to claim its NFT tokens from the orderbook (since the NFT token transfer would fail).

Recommendation: The following scenarios should be analysed for potentially compliance reasons:

- (3/4/5) Should buy/sell limit orders of banned users be able to be filled/matched?
- If no, it would require a heavy architectural change in the book keeping of the orderbook.
- If yes, perhaps these scenarios should be documented for the users.

Paintswap: Acknowledged. As you mentioned it would require too many changes to account for this and the orderbook is also not the only place that these could be traded or used (such as collateral for loans) as they are just standard NFTs so we cannot foresee and prevent all possible cases. Worst case all transfers can be paused while a solution is developed.

Cantina Managed: Acknowledged.

3.4 Informational

3.4.1 Dangling Comment

Severity: Informational

Context: SonicOrderBook.sol#L271-L272

Description:

```
// Buying transfer the NFTs to the taker
```

The above is a dangling comment from:

```
if (!isBuy) {
  ...
} else {
  // Buying transfer the NFTs to the taker --- < this comment
  _safeTransferNFTsFromUs(sender, order.tokenId, order.quantity);
}
```

Recommendation: It should be replaced by:

```
// Selling, transfer all NFTs to us
```

Paintswap: Fixed in PR 86.

Cantina Managed: Fix verified.