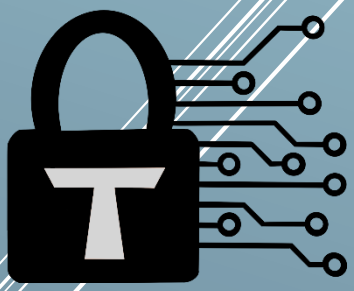
Trust Security

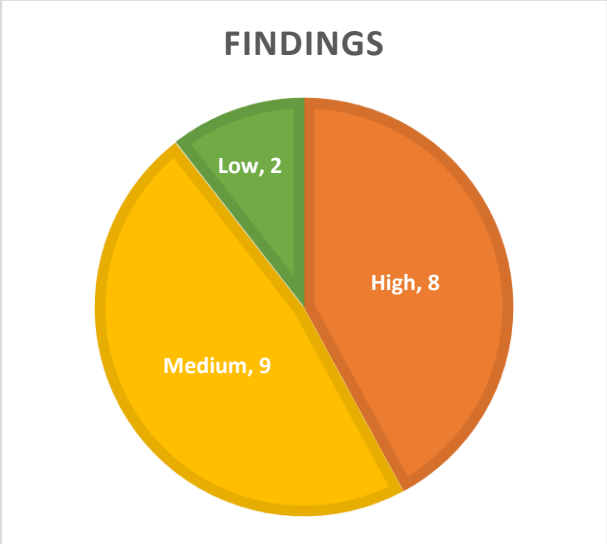


Smart Contract Audit

Clober V2 Protocol

28/02/24

Executive summary

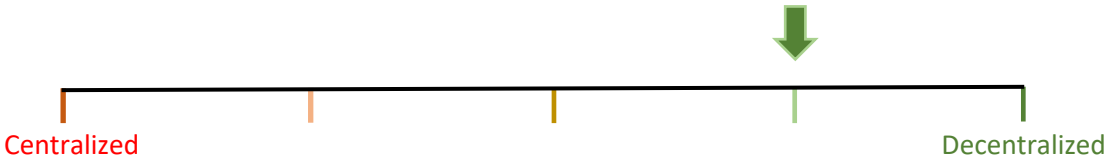


Category	Order Book DEX
Audited file count	15
Lines of Code	1620
Auditor	cccz HollaDieWaldfee
Time period	29/01/2024- 08/02/2024

Findings

Severity	Total	Fixed	Open	Acknowledged
High	8	8	-	-
Medium	9	9	-	-
Low	2	2	-	-

Centralization score



Signature

Document properties

Versioning

Version	Date	Description
0.1	08/02/2024	Client report
0.2	16/02/2024	Mitigation review
0.3	28/02/2024	Final mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

The following files are in scope of the audit:

- /contracts/Controller.sol
- /contracts/BookManager.sol
- /contracts/hooks/BaseHook.sol
- /contracts/hooks/BountyPlatform.sol
- /contracts/libraries/Book.sol
- /contracts/libraries/BookId.sol
- /contracts/libraries/FeePolicy.sol
- /contracts/libraries/Heap.sol
- /contracts/libraries/Hooks.sol
- /contracts/libraries/Lockers.sol
- /contracts/libraries/Math.sol
- /contracts/libraries/OrderId.sol
- /contracts/libraries/SignificantBit.sol
- /contracts/libraries/Tick.sol
- /contracts/libraries/TotalClaimableMap.sol

Repository details

- **Repository URL:** https://github.com/clober-dex/v2-core
- **Commit hash:** 3cc1fca8765ebeca54a6ef2add271e0db76726b4
- **Mitigation review hash:** 8fc51ac7d7f2111f16554f370e74d6f6e5619819

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

A top competitor in audit contests, cccz has achieved superstar status in the security space. He is a Black Hat / DEFCON speaker with rich experience in both traditional and blockchain security.

HollaDieWaldfee is a renowned security expert with a track record of multiple first places in competitive audits. He is a Lead Senior Watson at Sherlock and Senior Auditor for Trust Security and Renascence Labs.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed. Fuzz tests and unit tests have also been used as needed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, despite implementing custom data structures for efficiency.
Documentation	**Moderate**	Project is still under active development and currently lacks documentation.
Best practices	**Good**	Project generally follows best practices.
Centralization risks	**Good**	If parameters are chosen safely, there are no centralization risks, except for removing allowed providers.

Findings

High severity findings

TRST-H-1 BountyPlatform implements wrong balance check

- **Category:** Logical flaws
- **Source:** BountyPlatform.sol
- **Status:** Fixed

Description

The *BountyPlatform.afterMake()* function is called after an order has been made and allows to offer a bounty. To set a bounty, the bounty must be funded and so the *BountyPlatform* contract needs to check that its balance has at least increased by an amount that is equal to the bounty **amount**.

This check is faulty and instead of checking that the required **amount** has been added to the contract's balance, it is checked that the contract's balance is at least **amount**.

This allows an attacker to drain all funds from the *BountyPlatform* contract by posting a bounty that is equal to the contract's balance and then claiming the bounty.

Recommended mitigation

Instead of checking that **amount** is greater or equal to the *BountyPlatform*'s balance, **amount** should be checked against the difference between the contract's balance and its stored balance.

```
diff --git a/contracts/hooks/BountyPlatform.sol b/contracts/hooks/BountyPlatform.sol
index aed0e84..296ac82 100644
--- a/contracts/hooks/BountyPlatform.sol
+++ b/contracts/hooks/BountyPlatform.sol
@@ -43,7 +43,7 @@ contract BountyPlatform is BaseHook, Ownable2Step, IBountyPlatform {
             Bounty memory bounty = abi.decode(hookData, (Bounty));
             uint256 amount = _getAmount(bounty);
             if (amount > 0) {
-                if (bounty.currency.balanceOfSelf() < amount) revert
NotEnoughBalance();
+                if (bounty.currency.balanceOfSelf() - balance[bounty.currency] <
amount) revert NotEnoughBalance();
                 balance[bounty.currency] += amount;
                 _bountyMap[id] = bounty;
                 emit BountyOffered(id, bounty.currency, amount);
```

Team response

Fixed.

Mitigation Review

The recommendation has been implemented with a minor gas optimization.

TRST-H-2 BookManager._calculateAmountInReverse() calculation is incorrect

- **Category:** Logical flaws
- **Source:** BookManager.sol
- **Status:** Fixed

Description

The *BookManager._calculateAmountInReverse()* function is used in *BookManager.cancel()*. The function is needed to account for the fees the user has paid (positive maker fee rate) or the fees they have received (negative maker fee rate) when creating the order.

However, the calculation is incorrect. This can be easily recognized for positive maker fee rates as they cause an underflow which makes cancellation impossible.

Negative maker fee rates are even more impactful as they allow the user to receive more funds from cancellation than he has initially paid, effectively allowing him to drain the *BookManager*.

```
function _calculateAmountInReverse(uint256 amount, int24 rate) internal pure returns
(uint256 adjustedAmount) {
    uint256 fee = Math.divide(amount * uint256(_RATE_PRECISION),
uint256(_RATE_PRECISION - rate), rate < 0);
    adjustedAmount = rate > 0 ? amount - fee : amount + fee;
}
```

Assume $amount = 1000$ and $rate = -2000$ (-0.2%), we then have $fee = 1000 * 1e6/(1e6 + 2000) = 999$ (rounded up). Hence, $adjustedAmount = 1000 + 999 = 1999$. The loss to the protocol is $1999 - 998 = 1001$ with 998 being the correct reverse quote amount that accounts for the fee that has been paid out to the maker.

Recommended mitigation

We propose implementing the following formula instead.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..2874aa1 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -366,8 +366,13 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit
{
     }

     function _calculateAmountInReverse(uint256 amount, int24 rate) internal pure
returns (uint256 adjustedAmount) {
-        uint256 fee = Math.divide(amount * uint256(_RATE_PRECISION),
uint256(_RATE_PRECISION - rate), rate < 0);
-        adjustedAmount = rate > 0 ? amount - fee : amount + fee;
+        bool positive = rate > 0;
+        uint256 absRate;
+        unchecked {
+            absRate = uint256(uint24(positive ? rate : -rate));
+        }
+        uint256 absFee = Math.divide(amount * absRate, uint256(_RATE_PRECISION),
!positive);
+        adjustedAmount = positive ? amount + absFee : amount - absFee;
     }
```

First, the **absRate** is calculated. Based on this rate we calculate the **absFee** that has been paid or received for the **amount**. We round up the amount the maker must pay upon cancellation and round down the amount the maker receives upon cancellation. Finally, we add **absFee** to **amount** if the maker receives a refund of the fee and subtract **absFee** from **amount** if the maker has to pay back the fee.

Team response

Fixed.

Mitigation Review

The *_calculateAmountInReverse()* function has been fixed and renamed to *_calculateOriginalAmount()*. It has been recognized that the same behavior can be achieved by slightly modifying the *calculateFee()* function.

This means the *_calculateOriginalAmount()* function is currently never used and should be removed.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 00d6f8c..890ea20 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -358,16 +358,6 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit
{
         currencyDelta[locker][currency] = next;
     }

-    function _calculateOriginalAmount(uint256 amount, int24 rate) internal pure
returns (uint256 originalAmount) {
-        bool positive = rate > 0;
-        uint256 absRate;
-        unchecked {
-            absRate = uint256(uint24(positive ? rate : -rate));
-        }
-        uint256 absFee = Math.divide(amount * absRate,
FeePolicyLibrary.RATE_PRECISION, !positive);
-        originalAmount = positive ? amount + absFee : amount - absFee;
-    }
-
     function load(bytes32 slot) external view returns (bytes32 value) {
         assembly {
             value := sload(slot)
```

In addition, the calculation of the fee refund in *BookManager.cancel()* lacks a check whether the maker fee uses the **quote**. As a result, the fee is repaid regardless of whether it has been paid in *BookManager.make()*.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 00d6f8c..c295fea 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -226,8 +226,10 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit
{

         unchecked {
             canceledAmount = uint256(canceled) * key.unit;
-            int256 quoteFee = key.makerPolicy.calculateFee(canceledAmount, true);
-            canceledAmount = uint256(int256(canceledAmount) + quoteFee);
+            if (params.key.makerPolicy.usesQuote()) {
+                int256 quoteFee = key.makerPolicy.calculateFee(canceledAmount, true);
```

```
+                canceledAmount = uint256(int256(canceledAmount) + quoteFee);
+            }
         }

         if (pending == 0) _burn(OrderId.unwrap(params.id));
```

Team response

Fix1,Fix2.

Final Mitigation Review

The fix implements the recommendation.

TRST-H-3 Hook can bypass delta accounting and drain the BookManager

- **Category:** Reentrancy attacks
- **Source:** BookManager.sol
- **Status:** Fixed

Description

By executing the following sequence of actions, all funds in the *BookManager* can be stolen within a single transaction:

1. *open()* a new **Book** with a malicious **Hook** that has an *afterOpen()* callback
2. The *afterOpen()* callback is executed and **Hook** is set as the current hook in *Lockers.setCurrentHook()*
3. The **Hook** can now pass the *onlyByLocker* modifier
4. **Hook** withdraws all funds via *BookManager.withdraw()*
5. No **delta** checks are applied since the call to the **Hook** is not wrapped in a call to *BookManager.lock()*

Recommended mitigation

As the root cause it was identified that hooks must not be able to pass the *onlyByLocker* modifier in a context that is not wrapped in a *BookManager.lock()* call. There exist different options to fix this issue. The following two are likely the best options:

1. Add the *onlyByLocker* modifier to all functions that contain a hook callback.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..ae53a92 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -76,7 +76,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
         }
     }

-    function open(BookKey calldata key, bytes calldata hookData) external {
+    function open(BookKey calldata key, bytes calldata hookData) external
onlyByLocker {
         // @dev Also, the book opener should set unit at least circulatingTotalSupply
/ type(uint64).max to avoid overflow.
         //      But it is not checked here because it is not possible to check it
without knowing circulatingTotalSupply.
         if (key.unit == 0) revert InvalidUnit();
@@ -203,7 +203,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
```

```
         emit Take(bookId, msg.sender, tick, takenAmount);
     }

-    function cancel(CancelParams calldata params, bytes calldata hookData) external {
+    function cancel(CancelParams calldata params, bytes calldata hookData) external
onlyByLocker {
         address owner = _requireOwned(OrderId.unwrap(params.id));
         _checkAuthorized(owner, msg.sender, OrderId.unwrap(params.id));

@@ -233,7 +233,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
         emit Cancel(params.id, canceled);
     }

-    function claim(OrderId id, bytes calldata hookData) external {
+    function claim(OrderId id, bytes calldata hookData) external onlyByLocker {
         // @dev Load owner with nonexistent token check.
         //      We don't need to check the authorization because claiming another
user's order is allowed.
         address owner = _requireOwned(OrderId.unwrap(id));
```

2. Remove the ability for hooks to call *onlyByLocker* protected functions.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..192bdc9 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -58,7 +58,6 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
         address locker = Lockers.getCurrentLocker();
         IHooks hook = Lockers.getCurrentHook();
         if (caller == locker) return;
-        if (caller == address(hook) && hook.hasPermission(Hooks.ACCESS_LOCK_FLAG))
return;
         revert LockedBy(locker, address(hook));
     }
```

We recommend implementing option 1 rather than option 2 since it's the behavior implemented by Uniswap V4 (following the Uniswap V4 implementation makes development of Clober V2 periphery easier as concepts from Uniswap V4 can continue to be adopted).

However, implementing option 1 breaks the functionality in *Controller.claim()* and *Controller.cancel()*. Given that *Controller* contains a lot of issues and requires broader refactoring, this shouldn't be too much of a constraint.

Team response

Fixed.

Mitigation Review

The issue has been fixed as recommended by implementing option 1.

TRST-H-4 Controller.lockAcquired() consumes the tokens approved by users

- **Category:** Missing input validation
- **Source:** Controller.sol
- **Status:** Fixed

Description

In addition to the path *Controller.execute() -> BookManager.lock() -> Controller.lockAcquired()*, attack can call *BookManager.lock() -> Controller.lockAcquired()* directly, so that they can control the data in *Controller.lockAcquired().*

For users that have given approval to the *Controller*, the following attack is possible:

1. The attacker directly calls *BookManager.lock()*, where **locker** is the *Controller* contract, and data is made of the victim's address (e.g. unlimited approved user) and a low price (Sell 1 ETH for 0.1 USDC).
2. *Controller.lockAcquired()* will transfer tokens from the victim to create the order.
3. The attacker takes the order and profits from the price difference.

Recommended mitigation

It is recommended to use the **lockCaller** parameter of Controller.*lockAcquired()* and require that **lockCaller == address(this)**, i.e., the call is required to be initiated from the *Controller*.

```
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -69,8 +69,9 @@ contract Controller is IController, ILocker {
         return tick.toPrice();
     }

-    function lockAcquired(address, bytes memory data) external returns (bytes memory
returnData) {
+    function lockAcquired(address lockCaller, bytes memory data) external returns
(bytes memory returnData) {
         if (msg.sender != address(_bookManager)) revert InvalidAccess();
+        if (lockCaller != address(this)) revert InvalidAccess();
         (
             address user,
             Action[] memory actionList,
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-H-5 Controller._take() misses an increment of spendBaseAmount which bypasses the slippage check

- **Category:** Logical flaws
- **Source:** Controller.sol
- **Status:** Fixed

Description

The loop execution in *Controller._take()* is terminated when **leftQuoteAmount <= quoteAmount**.

```
    function _take(TakeOrderParams memory params) internal {
        IBookManager.BookKey memory key = _bookManager.getBookKey(params.id);
```

```
        uint256 leftQuoteAmount = params.quoteAmount;
        uint256 spendBaseAmount;

        uint256 quoteAmount;
        uint256 baseAmount;
        while (leftQuoteAmount > quoteAmount) {
            unchecked {
                leftQuoteAmount -= quoteAmount;
                spendBaseAmount += baseAmount;
            }
            (quoteAmount, baseAmount) = _bookManager.take(
                IBookManager.TakeParams({key: key, maxAmount:
leftQuoteAmount.divide(key.unit, true).toUint64()}),
                params.hookData
            );
            if (quoteAmount == 0) break;
        }
        if (params.maxBaseAmount < spendBaseAmount) revert ControllerSlippage();
    }
```

This means **spendBaseAmount** is not incremented after the last loop iteration, i.e., when all the remaining quote amount has been taken, leading to a wrong slippage check.

Assume the scenario when the entire quote amount is taken at the first tick. Then, **spendBaseAmount** is never incremented and is equal to zero. This completely bypasses the slippage check and can lead to an immediate loss to the user.

Recommended mitigation

The suggested fix for this specific problem is given below. Note that this needs to be integrated with the other fixes for the slippage control.

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index ece42c4..f31058a 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -215,13 +215,13 @@ contract Controller is IController, ILocker {
         while (leftQuoteAmount > quoteAmount) {
             unchecked {
                 leftQuoteAmount -= quoteAmount;
-                spendBaseAmount += baseAmount;
             }
             (quoteAmount, baseAmount) = _bookManager.take(
                 IBookManager.TakeParams({key: key, maxAmount:
leftQuoteAmount.divide(key.unit, true).toUint64()}),
                 params.hookData
             );
             if (quoteAmount == 0) break;
+            spendBaseAmount += baseAmount;
         }
         if (params.maxBaseAmount < spendBaseAmount) revert ControllerSlippage();
     }
```

Team response

Fixed.

Mitigation Review

The referenced PR implements the recommendation.

However, both the *Controller._take()* and *Controller._spend()* function have been heavily refactored in the latest commit. They use a different slippage control mechanism by checking that **limitPrice >= lowest tick**. Hence, the **spendBaseAmount** variable has been removed entirely and the issue has been fixed.

TRST-H-6 NFT can get stuck in Controller and claimed funds can be stolen

- **Category:** Logical flaws
- **Source:** Controller.sol
- **Status:** Fixed

Description

Before calling *BookManager.claim()* or *BookManager.cancel()*, the Controller transfers the NFT from the user to itself.

Hence it is critical that the NFT (in case it has not been burned) is transferred back to the user after the operation in the *BookManager* has been performed.

The problem is that the following check might fail when trying to transfer the NFT back.

```
try _bookManager.cancel(
    IBookManager.CancelParams({id: params.id, to: (params.leftQuoteAmount /
key.unit).toUint64()}),
    params.hookData
) {} catch {}
if (_bookManager.getOrder(params.id).claimable > 0 || params.leftQuoteAmount > 0) {
    _bookManager.transferFrom(address(this), user, orderId);
}
```

In particular, **claimable** might be zero and **leftQuoteAmount** might be zero, but since the cancellation has failed, **leftQuoteAmount=0** might not imply that **_bookManager.getOrder(params.id).open == 0**. This means the NFT is not transferred back even though it still represents an open order.

This scenario can occur in multiple ways. For example:

1. Attacker front-runs a cancellation for an order to take it partially, this may make the cancellation fail. They then call *BookManager.claim()* for the order such that the **claimable > 0** check does not apply.
2. The *beforeCancel()* hook returns early.
3. The attacker creates empty ticks with *Controller.make(0)* and makes the call to *cleanHeap()* revert.

Note that once the NFT is stuck in the *Controller* it cannot be recovered anymore and any user can claim its claimable funds and steal them from the *Controller.*

Recommended mitigation

The *Controller.cancel()* function should be fixed like so:

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
```

```
index ece42c4..e0f067b 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -280,7 +280,7 @@ contract Controller is IController, ILocker {
                IBookManager.CancelParams({id: params.id, to: (params.leftQuoteAmount /
key.unit).toUint64()}),
                params.hookData
            ) {} catch {}
-           if (_bookManager.getOrder(params.id).claimable > 0 || params.leftQuoteAmount
> 0) {
+           if (_bookManager.getOrder(params.id).claimable > 0 ||
_bookManager.getOrder(params.id).open > 0) {
                _bookManager.transferFrom(address(this), user, orderId);
            }
```

The same issue exists for *Controller.claim()* when the *beforeClaim()* hook returns early. We recommend the following fix:

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index ece42c4..d565a4d 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -262,7 +262,7 @@ contract Controller is IController, ILocker {
            if (_bookManager.getApproved(orderId) == address(this)) {
                _bookManager.transferFrom(user, address(this), orderId);
                _bookManager.claim(params.id, params.hookData);
-               if (_bookManager.getOrder(params.id).open > 0) {
+               if (_bookManager.getOrder(params.id).open > 0 ||
_bookManager.getOrder(params.id).claimable > 0) {
                    _bookManager.transferFrom(address(this), user, orderId);
                }
            } else {
```

Team response

Fixed.

Mitigation Review

The fix has moved the NFT transfers into the *execute()* function and the recommended check has been implemented. All NFTs that have been transferred to the *Controller* are now transferred back **if orderInfo.claimable > 0 || orderInfo.open > 0**, i.e., if the NFT still exists.

TRST-H-7 Fee accounting in BookManager is directionally wrong

- **Category:** Logical flaws
- **Source:** BookManager.sol
- **Status:** Fixed

Description

A positive maker / taker fee rate means the maker / taker has to pay the fee, while a negative maker / taker fee rate means the maker / taker is getting paid the fee. Therefore in *Controller.make()*, **quoteDelta** needs to be increased for positive fees and decreased for negative fees.

```
        if (!params.key.makerPolicy.useOutput()) {
            quoteDelta -= _calculateFee(quoteAmount, params.key.makerPolicy.rate());
```

```
        }
```

In *Controller.take()*, **baseDelta** also needs to be increased for positive fees and decreased for negative fees.

```
            } else {
                baseDelta -= _calculateFee(baseAmount, params.key.takerPolicy.rate());
            }
```

Recommended mitigation

The following fix will correct the fee accounting.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..774864c 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -157,7 +157,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
         int256 quoteDelta = quoteAmount.toInt256();

         if (!params.key.makerPolicy.useOutput()) {
-            quoteDelta -= _calculateFee(quoteAmount, params.key.makerPolicy.rate());
+            quoteDelta += _calculateFee(quoteAmount, params.key.makerPolicy.rate());
         }

         _accountDelta(params.key.quote, quoteDelta);
@@ -192,7 +192,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
             if (params.key.takerPolicy.useOutput()) {
                 quoteDelta -= _calculateFee(quoteAmount,
params.key.takerPolicy.rate());
             } else {
-                baseDelta -= _calculateFee(baseAmount,
params.key.takerPolicy.rate());
+                baseDelta += _calculateFee(baseAmount,
params.key.takerPolicy.rate());
             }
             _accountDelta(params.key.quote, -quoteDelta);
             _accountDelta(params.key.base, baseDelta);
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-H-8 Controller slippage control is bypassed for partially filled orders

- **Category:** Logical flaws
- **Source:** Controller.sol
- **Status:** Fixed

Description

The slippage control in *Controller._take()* is implemented by checking that **spendBaseAmount <= params.maxBaseAmount**, while in *Controller._spend()* it is implemented by checking that **takenQuoteAmount >= params.minQuoteAmount**.

There are multiple ways in which these checks can be bypassed, one of these can even be controlled by an attacker. The root cause is that implementing slippage control in this way doesn't account for partially filled orders.

Bypass 1**:**

BookManager.take() can return early due to the hook.

As a result, *Controller._take()* breaks out of the loop.

This can make the slippage check pass even though for the part of the quote amount that could be matched, the price might have been a lot higher than expected.

In *Controller._spend()* the slippage control would just end up being more restrictive.

Bypass 2:

An empty tick causes *Controller._take()* to return early. If the lowest tick is empty, then depth is 0, and **quoteAmount=0**

An attacker can call *BookManager.make(0)* to insert a tick with zero depth.

If the victim calls *Controller._take()* and the amount they take makes the price reach the zero depth tick this means the slippage control is bypassed.

The impact is the same as in Bypass 1.

Bypass 3:

leftQuoteAmount may be rounded down to zero due to division by **key.unit**.

If the total value of **params.quoteAmount** was very low (say 1.5 units), this can make a big difference in terms of slippage.

Recommended mitigation

There are two recommended ways to fix the issue.

1) Revert if the **limitPrice** is exceeded.

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index ece42c4..81a2b05 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -213,6 +213,7 @@ contract Controller is IController, ILocker {
         uint256 quoteAmount;
         uint256 baseAmount;
         while (leftQuoteAmount > quoteAmount) {
+            if (params.limitPrice < _bookManager.getRoot(params.id).toPrice()) revert
ControllerSlippage();
             unchecked {
                 leftQuoteAmount -= quoteAmount;
                 spendBaseAmount += baseAmount;
@@ -234,6 +235,7 @@ contract Controller is IController, ILocker {

         while (leftBaseAmount > 0 && !_bookManager.isEmpty(params.id)) {
             Tick tick = _bookManager.getRoot(params.id);
```

```
+            if (params.limitPrice < tick.toPrice()) revert ControllerSlippage();
             (uint256 quoteAmount, uint256 baseAmount) = _bookManager.take(
                 IBookManager.TakeParams({
                     key: key,
```

2) Calculate an average execution price at the end of the function and check that it's **<= limitPrice**.

In both cases the **limitPrice** variable which already exists in the **TakeOrderParams** and **SpendOrderParams** structs can be used for this check.

Team response

Fixed.

Mitigation Review

The first suggestion has been implemented with regards to using a **limitPrice** and the slippage control now works for partially filled orders.

Instead of reverting if the **limitPrice** is exceeded, the function breaks. This is just a convention and does not impact the effectiveness of the slippage control.

Medium severity findings

TRST-M-1 Controller._settle() can be griefed by out-of-sync reservesOf

- **Category:** Griefing attacks
- **Source:** Controller.sol
- **Status:** Fixed

Description

Once the call to *Controller.lockAcquired()* is over, execution is resumed in *BookManager.lock()* and it is checked that **nonzeroDeltaCount=0**.

To resolve a positive **delta**, the *Controller._settle()* function simply sends the amounts of funds equal to the size of the **delta** but it does not account for the funds that have already been sent to the *BookManager.*

```
    function settle(Currency currency) external payable onlyByLocker returns (uint256
paid) {
        uint256 reservesBefore = reservesOf[currency];
        reservesOf[currency] = currency.balanceOfSelf();
        paid = reservesOf[currency] - reservesBefore;
        // subtraction must be safe
        _accountDelta(currency, -(paid.toInt256()));
    }
```

If a griefer has sent an arbitrarily small amount of funds, the **delta** becomes negative. This means that the **nonzeroDeltaCount=0** check in *BookManager.lock()* fails, and the transaction reverts.

Recommended mitigation

It is recommended to simply try to cover the positive **currencyDelta**. If there are additional tokens in the *BookManager* it would withdraw them again.

This is the most efficient solution in almost all circumstances.

```
    function _settleTokens(address user, ERC20PermitParams[] memory relatedTokenList)
internal {
        uint256 length = relatedTokenList.length;
        _permitERC20(relatedTokenList);
        Currency native = CurrencyLibrary.NATIVE;
        int256 currencyDelta = _bookManager.currencyDelta(address(this), native);
        if (currencyDelta > 0) {
            native.transfer(address(_bookManager), uint256(currencyDelta));
            _bookManager.settle(native);
+       }
+       int256 currencyDelta = _bookManager.currencyDelta(address(this), native);
+       if (currencyDelta < 0) {
-       } else if (currencyDelta < 0) {
            _bookManager.withdraw(CurrencyLibrary.NATIVE, user, uint256(-
currencyDelta));
        }
        for (uint256 i = 0; i < length; ++i) {
            Currency currency = Currency.wrap(relatedTokenList[i].token);
            currencyDelta = _bookManager.currencyDelta(address(this), currency);
            if (currencyDelta > 0) {
                IERC20(relatedTokenList[i].token).safeTransferFrom(user,
address(_bookManager), uint256(currencyDelta));
                _bookManager.settle(currency);
```

```
+           }
+           int256 currencyDelta = _bookManager.currencyDelta(address(this), native);
+           if (currencyDelta < 0) {
-           } else if (currencyDelta < 0) {
                _bookManager.withdraw(Currency.wrap(relatedTokenList[i].token), user,
uint256(-currencyDelta));
            }
            uint256 balance =
IERC20(relatedTokenList[i].token).balanceOf(address(this));
            if (balance > 0) {
                IERC20(relatedTokenList[i].token).transfer(user, balance);
            }
        }
        if (address(this).balance > 0) native.transfer(user, address(this).balance);
    }
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-M-2 Incorrect implementation of Controller._permitERC20()

- **Category:** Logical flaws
- **Source:** Controller.sol
- **Status:** Fixed

Description

Controller._permitERC20() uses signatures to approve *Controller* to consume user tokens, where the **owner** parameter is currently set to **msg.sender**.

```
    function _permitERC20(ERC20PermitParams[] memory permitParamsList) internal {
        uint256 length = permitParamsList.length;
        for (uint256 i = 0; i < length; ++i) {
            ERC20PermitParams memory permitParams = permitParamsList[i];
            if (permitParams.signature.deadline > 0) {
                try IERC20Permit(permitParams.token).permit(
                    msg.sender,
                    address(this),
...
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual {
```

However, notice that *lockAcquired()* calls *_settleTokens()* and *_settleTokens()* calls *_permitERC20()*. Since *lockAcquired()* is called by **_bookManager**, it means that the **msg.sender** in *_permitERC20()* is **_bookManager**, not the **user**.

```
    function lockAcquired(address, bytes memory data) external returns (bytes memory
returnData) {
        if (msg.sender != address(_bookManager)) revert InvalidAccess();
        ...
        _settleTokens(user, relatedTokenList);
...
    function _settleTokens(address user, ERC20PermitParams[] memory relatedTokenList)
internal {
        uint256 length = relatedTokenList.length;
        _permitERC20(relatedTokenList);
```

In the worst case scenario, since the signature here won't be used, a malicious user can steal the signature.

Combined with TRST-H-4, there is a risk of signature abuse. Note that TRST-H-4 makes ERC20 signatures fundamentally incompatible since they can be front-run. This finding deals with the fact that calls to *permit()* currently revert due to the incorrect parameter.

ERC20 signature abuse:

1. Alice signs to approve the *Controller* to use her tokens, calls *execute()* with the signature as parameter to create an order, however the transaction fails due to the error in *_permitERC20()*.
2. Bob steals Alice's signature from the transaction and manually calls *BookManager.permit()* to approve the *Controller* to use Alice's tokens.
3. Bob calls *BookManager.lock(address(Controller),data)*, constructs the data using Alice's address and malicious data (say a low price), and makes **permitParams.deadline = 0**. After this in *_settleTokens()*, the *_permitERC20()* function will just return.
4. *BookManager.lock()* calls the *Controller.lockAcquired()* function and then transfers tokens from Alice to settle the delta for the order.
5. Bob takes Alice's order at a low price.

Recommended mitigation

It is recommended to add the **user** parameter to *_permitERC20()* and use **user** instead of **msg.sender** as the **owner** when calling *permit()*.

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index ece42c4..7da5fa5 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -287,7 +287,7 @@ contract Controller is IController, ILocker {

     function _settleTokens(address user, ERC20PermitParams[] memory relatedTokenList)
internal {
         uint256 length = relatedTokenList.length;
-        _permitERC20(relatedTokenList);
+        _permitERC20(relatedTokenList, user);
         Currency native = CurrencyLibrary.NATIVE;
         int256 currencyDelta = _bookManager.currencyDelta(address(this), native);
         if (currencyDelta > 0) {
@@ -313,13 +313,13 @@ contract Controller is IController, ILocker {
         if (address(this).balance > 0) native.transfer(user, address(this).balance);
     }

-    function _permitERC20(ERC20PermitParams[] memory permitParamsList) internal {
+    function _permitERC20(ERC20PermitParams[] memory permitParamsList, address user)
internal {
```

```
         uint256 length = permitParamsList.length;
         for (uint256 i = 0; i < length; ++i) {
             ERC20PermitParams memory permitParams = permitParamsList[i];
             if (permitParams.signature.deadline > 0) {
                 try IERC20Permit(permitParams.token).permit(
-                    msg.sender,
+                    user,
                     address(this),
                     permitParams.permitAmount,
                     permitParams.signature.deadline,
```

Team response

Fixed.

Mitigation Review

Controller_permitERC20() is now called outside of *BookManager.lock()*, in a context where **msg.sender** is the user, not the *BookManager*. This means the recommended change is not necessary and the issue is fixed.

TRST-M-3 Incorrect implementation of Controller._permitERC721()

- **Category:** Logical flaws
- **Source:** Controller.sol
- **Status:** Fixed

Description

Controller._permitERC721() uses signatures to approve *Controller* to use the user's NFT, but the **spender** here is **msg.sender** and not *Controller*.

```
    function _permitERC721(uint256 tokenId, PermitSignature memory permitParams)
internal {
        if (permitParams.deadline > 0) {
            try IERC721Permit(address(_bookManager)).permit(
                msg.sender, tokenId, permitParams.deadline, permitParams.v,
permitParams.r, permitParams.s
            ) {} catch {}
        }
    }
...
    function permit(address spender, uint256 tokenId, uint256 deadline, uint8 v,
bytes32 r, bytes32 s)
        external
        override
    {
```

Again, combined with TRUST-H-4 there is an increased risk of signature abuse:

1. Alice creates a signature to approve the *Controller* to use her NFT, calls *execute()* with the signature as parameter to cancel an order partially from amount 100 to amount 50 , however the transaction fails due to the error in *_permitERC721()*.
2. Bob steals Alice's signature from the transaction and manually calls *BookManager.permit()* to approve the *Controller* to use Alice's NFT.

3. Bob calls *BookManager.lock(address(Controller),data)*, and constructs the data using Alice's address and malicious data (cancel from 100 to 0).
4. *BookManager.lock()* will call the *Controller.lockAcquired()* function and then cancel Alice's order from 100 to 0 but Alice's expectation is from 100 to 50.

Recommended mitigation

It is recommended to use **address(this)** instead of **msg.sender** as the **spender** when calling *permit()*.

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index ece42c4..55da128 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -334,7 +334,7 @@ contract Controller is IController, ILocker {
     function _permitERC721(uint256 tokenId, PermitSignature memory permitParams)
internal {
         if (permitParams.deadline > 0) {
             try IERC721Permit(address(_bookManager)).permit(
-                msg.sender, tokenId, permitParams.deadline, permitParams.v,
permitParams.r, permitParams.s
+                address(this), tokenId, permitParams.deadline, permitParams.v,
permitParams.r, permitParams.s
             ) {} catch {}
         }
     }
```

Team response

Fixed.

Mitigation Review

The issue still exists. It is still needed to set **spender=address(this)** instead of **msg.sender**.

The downstream functions in the *BookManager* require that the *Controller* is approved, not the user that has called the *Controller*.

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index bc1b878..ad27403 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -385,7 +385,7 @@ contract Controller is IController, ILocker, ReentrancyGuard {
             PermitSignature memory signature = permitParamsList[i].signature;
             if (signature.deadline > 0) {
                 try IERC721Permit(address(_bookManager)).permit(
-                    msg.sender, permitParamsList[i].tokenId, signature.deadline,
signature.v, signature.r, signature.s
+                    address(this), permitParamsList[i].tokenId, signature.deadline,
signature.v, signature.r, signature.s
                 ) {} catch {}
             }
         }
```

Team response

Fixed.

Final Mitigation Review

The fix implements the recommendation.

TRST-M-4 BookManager.collect() is missing reservesOf accounting

- **Category:** Logical flaws
- **Source:** BookManager.sol
- **Status:** Fixed

Description

The **reservesOf** accounting is missing from the *BookManager.collect()* function.

Consider the following scenario:

1. Maker fee = Taker fee = 2%.
2. Alice calls *BookManager.make()* with **quoteAmount=1000**, **quoteFee=20**. In *BookManager.settle()* it sets **reservesOf[quote]=1020**.
3. Bob calls *BookManager.take()* with **takenAmount=100** and **baseFee=2**. In *BookManager.withdraw()* it sets **reservesOf[quote]=1020-1000=20**. In *BookManager.settle()* it sets **reservesOf[base]=102**.
4. Alice calls *BookManager.claim()* and sets **reservesOf[base] = 2**.
5. **provider** collects the fees, but **reservesOf[base]** is still 2, **reservesOf[quote]** is still 20

As a result of this incorrect accounting, once users need to call *BookManager.settle()* again to settle a positive delta, they need to overpay.

Recommended mitigation

It is recommended to add the missing accounting step in *BookManager.collect()*.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..c01c745 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -296,6 +296,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {

     function collect(address provider, Currency currency) external {
         uint256 amount = tokenOwed[provider][currency];
         if (amount > 0) {
             tokenOwed[provider][currency] = 0;
+            reservesOf[currency] -= amount;
             currency.transfer(provider, amount);
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-M-5 DoS in cleanHeap() blocks take() and cancel()

- **Category:** Griefing attacks
- **Source:** BookManager.sol

- **Status:** Fixed

Description

An attacker can make an order with a zero amount.

```
    function make(MakeParams calldata params, bytes calldata hookData)
        external
        onlyByLocker
        returns (OrderId id, uint256 quoteAmount)
    {
        if (params.provider != address(0) && !isWhitelisted[params.provider]) revert
InvalidProvider(params.provider);
        params.tick.validate();
        BookId bookId = params.key.toId();
        Book.State storage book = _books[bookId];
        book.checkOpened();

        if (!params.key.hooks.beforeMake(params, hookData)) return (OrderId.wrap(0),
0);

        uint40 orderIndex = book.make(params.tick, params.amount, params.provider);
```

In *Book.make(),* this adds an empty tick to heap.

```
    function make(State storage self, Tick tick, uint64 amount, address provider)
        internal
        returns (uint40 orderIndex)
    {
        uint24 tickIndex = tick.toUint24();
        if (!self.heap.has(tickIndex)) self.heap.push(tickIndex);
```

Attacker can call *make()* to push a lot of empty ticks into the heap, *cleanHeap()* will call *heap.pop()* several times in the loop, and when there are enough empty ticks, *cleanHeap()* will revert due to exceeding the block gas limit.

```
    function cleanHeap(State storage self) internal {
        while (!self.heap.isEmpty()) {
            if (depth(self, self.heap.root().toTick()) == 0) self.heap.pop();
            else break;
        }
    }
```

Since *cleanHeap()* will be called in *cancel()* and *take()*, this will make it impossible for the users to cancel and take orders. The DoS is only temporary since it can be resolved by creating a make order with a non-zero amount such that an empty tick becomes a non-empty tick.

Recommended mitigation

It is recommended that only orders with **amount > 0** can be created.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..507d823 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -141,6 +141,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
         returns (OrderId id, uint256 quoteAmount)
     {
         if (params.provider != address(0) && !isWhitelisted[params.provider]) revert
InvalidProvider(params.provider);
```

```
+         if (params.amount = 0) revert InvalidAmount();
          params.tick.validate();
          BookId bookId = params.key.toId();
          Book.State storage book = _books[bookId];
```

Team response

Fixed.

Mitigation Review

The missing zero amount check has been added to *Book.make()*, no empty orders can be created.

TRST-M-6 In Controller.cancel(), ERC721 permit approvals can be front-run

- **Category:** Front-running
- **Source:** Controller.sol
- **Status:** Fixed

Description

The *BookManager.cancel()* function checks that **msg.sender** is authorized for the order NFT.

```
function cancel(CancelParams calldata params, bytes calldata hookData) external {
    address owner = _requireOwned(OrderId.unwrap(params.id));
    _checkAuthorized(owner, msg.sender, OrderId.unwrap(params.id));
```

This is a problem if the *Controller* is approved via *ERC721.permit()* signatures or if regular approval and execution of *Controller.cancel()* is split into two transactions.

In the case of the permit, an attacker can use the signature to front-run the legitimate transaction, call *ERC721.permit()* with the signature to authorize the *Controller* to handle the NFT and then call *Controller.cancel()* to cancel a different amount from what is intended.

Since signatures can be passed to *Controller.cancel()* and *Controller.claim()* without being used, this increases the risk of signature abuse.

Recommended mitigation

The solution is to check in *Controller.cancel()* that the **msg.sender** is authorized for the order NFT, essentially:

- **msg.sender** is the **owner** or
- **msg.sender** has operator approval from **owner** or
- **msg.sender** has approval from **owner** for the specific NFT

In order to achieve this, the *BookManager* should publicly expose the *checkAuthorized()* function and it should be called in *Controller.cancel()*.

The parameters of *Controller.cancel()* and *Controller.claim()* should be modified to remove the signature.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
```

```
index 4965f8c..47e2195 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -62,6 +62,10 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
         revert LockedBy(locker, address(hook));
     }

+    function checkAuthorized(address owner, address spender, uint256 tokenId)
external view {
+        _checkAuthorized(owner, spender, tokenId);
+    }
+
     function getBookKey(BookId id) external view returns (BookKey memory) {
         return _books[id].key;
     }
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index ece42c4..af3dc00 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -167,23 +167,22 @@ contract Controller is IController, ILocker {
         _bookManager.lock(address(this), lockData);
     }

-    function claim(ClaimOrderParams[] calldata orderParamsList, uint64 deadline)
external checkDeadline(deadline) {
-        uint256 length = orderParamsList.length;
+    function claim(OrderId[] ids, bytes[] hookData, uint64 deadline) external
checkDeadline(deadline) {
+        uint256 length = ids.length;
         for (uint256 i = 0; i < length; ++i) {
-            ClaimOrderParams memory params = orderParamsList[i];
-            _bookManager.claim(params.id, params.hookData);
+            _bookManager.claim(ids[i], hookData[i]);
         }
     }

-    function cancel(CancelOrderParams[] calldata orderParamsList, uint64 deadline)
external checkDeadline(deadline) {
-        uint256 length = orderParamsList.length;
+    function cancel(OrderId[] ids, uint256[] leftQuoteAmounts, bytes[] hookData,
uint64 deadline) external checkDeadline(deadline) {
+        uint256 length = ids.length;
         for (uint256 i = 0; i < length; ++i) {
-            CancelOrderParams memory params = orderParamsList[i];
-            (BookId bookId,,) = params.id.decode();
+            _bookManager.checkAuthorized(_bookManager.ownerOf(ids[i]), msg.sender,
ids[i]);
+            (BookId bookId,,) = ids[i].decode();
             IBookManager.BookKey memory key = _bookManager.getBookKey(bookId);
             try _bookManager.cancel(
-                IBookManager.CancelParams({id: params.id, to: (params.leftQuoteAmount
/ key.unit).toUint64()}),
-                params.hookData
+                IBookManager.CancelParams({id: ids[i], to: (leftQuoteAmounts[i] /
key.unit).toUint64()}),
+                hookData[i]
             ) {} catch {}
         }
     }
```

Team response

Fixed.

Mitigation Review

The authorization check in *Controller.execute()* is applied to the ERC721s in the **ERC20PermitParams[]** array, but the actual NFTs that actions are performed on are encoded in the **paramsDataList.** Therefore, the authorization check needs to be moved into *Controller._cancel()* and *Controller._claim().*

The attacker can steal the signature and provide an empty **erc721PermitParamsList** in *execute()*, then claim or cancel the order and steal the received tokens.

Note that only *Controller.execute()* is vulnerable. *Controller.cancel()* and *Controller.claim()* are safe.

Team response

Fixed.

Final Mitigation Review

The fix moves the ERC721 authorization check to *lockAcquired()* to solve the issue.

TRST-M-7 Controller slippage check does not consider fees

- **Category:** Logical flaws
- **Source:** Controller.sol
- **Status:** Fixed

Description

The slippage checks in *Controller._take()* and *Controller._spend()* perform the slippage checks based on the **quoteAmount** and **baseAmount** that are returned by the *BookManager.take()* function.

These values do not include fees and so the effective order prices may be worse than what is indicated by the **quoteAmount** and **baseAmount**.

Recommended mitigation

There are two options to fix this problem:

1. Calculate the fee in the *Controller*
2. Return the amounts with the fees included in *BookManager.make()* and *BookManager.take()*

The second option seems more efficient even though it requires the change to be a layer deeper in the *BookManager* and there may be valid reasons for the amounts to *not* include fees.

The fix with Option 2 is suggested below. Note that the direction of the fee accounting is also fixed in this code snippet.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..ad21b66 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -138,7 +138,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
     function make(MakeParams calldata params, bytes calldata hookData)
         external
```

```
         onlyByLocker
-        returns (OrderId id, uint256 quoteAmount)
+        returns (OrderId id, uint256 absQuoteAmount)
     {
         if (params.provider != address(0) && !isWhitelisted[params.provider]) revert
InvalidProvider(params.provider);
         params.tick.validate();
@@ -152,13 +152,14 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit
{
         id = OrderIdLibrary.encode(bookId, params.tick, orderIndex);
         unchecked {
             // @dev uint64 * uint64 < type(uint256).max
-            quoteAmount = uint256(params.amount) * params.key.unit;
+            uint256 quoteAmount = uint256(params.amount) * params.key.unit;
         }
         int256 quoteDelta = quoteAmount.toInt256();

         if (!params.key.makerPolicy.useOutput()) {
-            quoteDelta -= _calculateFee(quoteAmount, params.key.makerPolicy.rate());
+            quoteDelta += _calculateFee(quoteAmount, params.key.makerPolicy.rate());
         }
+        absQuoteAmount = uint256(quoteDelta);

         _accountDelta(params.key.quote, quoteDelta);

@@ -172,7 +173,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
     function take(TakeParams calldata params, bytes calldata hookData)
         external
         onlyByLocker
-        returns (uint256 quoteAmount, uint256 baseAmount)
+        returns (uint256 absQuoteDelta, uint256 absBaseDelta)
     {
         BookId bookId = params.key.toId();
         Book.State storage book = _books[bookId];
@@ -182,9 +183,9 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {

         (Tick tick, uint64 takenAmount) = book.take(params.maxAmount);
         unchecked {
-            quoteAmount = uint256(takenAmount) * params.key.unit;
+            uint256 quoteAmount = uint256(takenAmount) * params.key.unit;
         }
-        baseAmount = tick.quoteToBase(quoteAmount, true);
+        uint256 baseAmount = tick.quoteToBase(quoteAmount, true);

         {
             int256 quoteDelta = quoteAmount.toInt256();
@@ -192,7 +193,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
             if (params.key.takerPolicy.useOutput()) {
                 quoteDelta -= _calculateFee(quoteAmount,
params.key.takerPolicy.rate());
             } else {
-                baseDelta -= _calculateFee(baseAmount,
params.key.takerPolicy.rate());
+                baseDelta += _calculateFee(baseAmount,
params.key.takerPolicy.rate());
             }
             _accountDelta(params.key.quote, -quoteDelta);
             _accountDelta(params.key.base, baseDelta);
@@ -200,6 +201,8 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {

         params.key.hooks.afterTake(params, tick, takenAmount, hookData);

+        absQuoteDelta = uint256(quoteDelta);
+        absBaseDelta = uint256(baseDelta);
         emit Take(bookId, msg.sender, tick, takenAmount);
     }
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation. The *BookManager* functions now return the amounts with the fees included. However, the **maxAmount** parameter that is passed to *_bookManager.take()* is incorrect, it should consider fees. Similarly, in *Controller_make(),* the **amount** parameter that is passed to *_bookManager.make()* should also consider fees.

Controller._make():

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index bc1b878..2d3163e 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -258,11 +258,14 @@ contract Controller is IController, ILocker, ReentrancyGuard {
     function _make(MakeOrderParams memory params) internal returns (OrderId id) {
         IBookManager.BookKey memory key = _bookManager.getBookKey(params.id);
         uint256 quoteAmount;
+        uint64 _amount = (params.quoteAmount / key.unit).toUint64();
         (id, quoteAmount) = _bookManager.make(
             IBookManager.MakeParams({
                 key: key,
                 tick: params.tick,
-                amount: (params.quoteAmount / key.unit).toUint64(),
+                amount: key.makerPolicy.usesQuote()
+                    ? (uint256(_amount) * 10**6).divide(uint256(int256(10 ** 6) +
key.makerPolicy.rate()), false).toUint64()
+                    : _amount,
                 provider: address(0)
             }),
             params.hookData
```

Controller._take():

```
@@ -282,11 +285,14 @@ contract Controller is IController, ILocker, ReentrancyGuard {
             unchecked {
                 leftQuoteAmount -= quoteAmount;
             }
+            uint64 _maxAmount = leftQuoteAmount.divide(key.unit, true).toUint64();
             (quoteAmount,) = _bookManager.take(
                 IBookManager.TakeParams({
                     key: key,
                     tick: tick,
-                    maxAmount: leftQuoteAmount.divide(key.unit, true).toUint64()
+                    maxAmount: key.takerPolicy.usesQuote()
+                    ? (uint256(_maxAmount) * 10**6).divide(uint256(int256(10 ** 6) -
key.takerPolicy.rate()), true).toUint64()
+                    : _maxAmount
                 }),
                 params.hookData
             );
```

Controller._spend():

```
@@ -299,22 +305,26 @@ contract Controller is IController, ILocker, ReentrancyGuard {

         uint256 leftBaseAmount = params.baseAmount;

-        while (leftBaseAmount > 0 && !_bookManager.isEmpty(params.id)) {
```

```
+        uint256 baseAmount;
+        while (leftBaseAmount > baseAmount && !_bookManager.isEmpty(params.id)) {
             Tick tick = _bookManager.getLowest(params.id);
             if (params.limitPrice < tick.toPrice()) break;
-            (, uint256 baseAmount) = _bookManager.take(
+            unchecked {
+                leftBaseAmount -= baseAmount;
+            }
+            uint64 _maxAmount = (tick.baseToQuote(leftBaseAmount, true) /
key.unit).toUint64();
+            (,baseAmount) = _bookManager.take(
                 IBookManager.TakeParams({
                     key: key,
                     tick: tick,
-                    maxAmount: (tick.baseToQuote(leftBaseAmount, false) /
key.unit).toUint64()
+                    maxAmount: !key.takerPolicy.usesQuote()
+                    ? (uint256(_maxAmount) * 10**6 ).divide(uint256(int256(10 ** 6) +
key.takerPolicy.rate()), true).toUint64()
+                    : _maxAmount
                 }),
                 params.hookData
             );
             if (baseAmount == 0) break;

-            unchecked {
-                leftBaseAmount -= baseAmount;
-            }
         }
     }
```

The formulas have been mathematically derived and it has been ensured that the code compiles. Still, the client is encouraged to provide greater test coverage for non-zero maker and taker fees in the *Controller*. No such tests exist as of now.

Team response

Fixed.

Final Mitigation Review

The fix implements the recommended logic. The rounding directions in the *Controller._make()*, *Controller._take()* and *Controller._spend()* functions are not in line with the recommendation but they are not of importance from a security perspective.

TRST-M-8 Book.cancel() needs to remove empty ticks

- **Category:** Griefing attacks
- **Source:** Book.sol
- **Status:** Fixed

Description

When *Book.cancel()* is called to cancel an order, it just pops empty ticks from the root of the heap. If the user creates a non-empty order at a tick that is not at the root of the heap and then fully cancels it with **to=0**, *Book.cancel()* will not remove that tick from the heap, resulting in empty ticks being left in the heap.

A malicious user could exploit this to create many empty ticks and thereby DoS *Book.cleanHeap()*. This leads to a DoS in the upstream *Book.take()* and *Book.cancel()* functions.

The impact is the same as in TRST-M-5 but the root cause is different.

Recommended mitigation

It is recommended to implement *Heap.remove()* as follows to allow removing ticks from any position in the heap.

```
diff --git a/contracts/libraries/Book.sol b/contracts/libraries/Book.sol
index 5a14bc6..91bf2cb 100644
--- a/contracts/libraries/Book.sol
+++ b/contracts/libraries/Book.sol
@@ -150,7 +150,14 @@ library Book {
         }
         queue.orders[orderIndex].pending = afterPending;

-        self.cleanHeap();
+        if (depth(self, tick) == 0) {
+            // remove() won't revert so we can cancel with to=0 even if the depth()
is already zero
+            // works even if heap is empty
+            self.heap.remove(tick.toUint24());
+        }
+        // we don't need to call cleanHeap() as calls to take() will take care of
removing empty ticks from the root
     }

     function cleanHeap(State storage self) internal {
diff --git a/contracts/libraries/Heap.sol b/contracts/libraries/Heap.sol
index 89bc29d..71e422f 100644
--- a/contracts/libraries/Heap.sol
+++ b/contracts/libraries/Heap.sol
@@ -82,6 +82,25 @@ library Heap {
         }
     }

+    function remove(mapping(uint256 => uint256) storage heap, uint24 value) internal
{
+        (uint256 b0b1, uint256 b2) = _split(value);
+        uint256 mask = 1 << b2;
+        uint256 b2Bitmap = heap[b0b1];
+
+        heap[b0b1] = b2Bitmap & (~mask);
+        if (b2Bitmap == mask) {
+            mask = 1 << (b0b1 & 0xff);
+            uint256 b1BitmapKey = ~(b0b1 >> 8);
+            uint256 b1Bitmap = heap[b1BitmapKey];
+
+            heap[b1BitmapKey] = b1Bitmap & (~mask);
+            if (mask == b1Bitmap) {
+                mask = 1 << (~b1BitmapKey);
+                heap[B0_BITMAP_KEY] = heap[B0_BITMAP_KEY] & (~mask);
+            }
+        }
+    }
+
     function minGreaterThan(mapping(uint256 => uint256) storage heap, uint24 value)
internal view returns (uint24) {
         (uint256 b0b1, uint256 b2) = _split(value);
         uint256 b2Bitmap = (MAX_UINT_256_MINUS_1 << b2) & heap[b0b1];
```

Team response

Fixed.

Mitigation Review

The new function is called *clear()* instead of *remove()*. So the comment should be corrected.

Apart from this comment, the issue has been fixed as recommended. *Book.cancel()* now removes ticks with empty liquidity.

Since *Book.take()* can now take liquidity from any tick, *TickBitmap.clear()* is used in *Book.take()* instead of *pop()*. In fact the *pop()* function is no longer needed and has been removed.

Team response

Fixed.

Final Mitigation Review

The fix corrects the comment.

TRST-M-9 Controller reentrancy due to ETH callback

- **Category:** Reentrancy attacks
- **Source:** Controller.sol
- **Status:** Fixed

Description

The protocol has different places where reentrancies can originate from. All of these are the result of ETH transfers.

Here are the sources for such reentrancies where the address that gets the callback is not necessarily the user that has initiated the whole transaction, which can lead to an exploit:

- *BookManager.cancel()*
- *BookManager.claim()*
- *BookManager.collect()*
- *BookManager.withdraw()*
- *BountyPlatform.afterClaim()*
- *BountyPlatform.afterCancel()*

A concrete way has not yet been discovered for the user getting the callback to himself managing to exploit the protocol. However this could still result in a viable attack in the future when more integrations are added. Adding additional *Hooks* and *Lockers* is non-trivial and can lead to complex vulnerabilities down the line.

Currently, the only possible uncovered attack is when a legitimate user (Bob) gives another user (Alice) a callback with the transaction going through *Controller.lockAcquired()*. Any of the above callbacks can be used in this attack.

The attack looks like this:

1. Alice tells Bob to claim her order via the *Controller.execute()* function, Bob might make this part of a larger sequence of actions (e.g. one of them might be to make an order for himself)
2. Bob rightfully thinks this is a safe action (in the base protocol you can just claim for any other user)
3. Alice gets a callback since the claimed ETH is sent to her.
4. Alice reenters the *Controller.execute()* function, acquires another lock and takes on positive delta.
5. Alice does not need to resolve the delta since the number of lockers is greater than one.
6. Bob will automatically resolve the delta (since delta is accounted per locker and currency) if he has given sufficient approval and calls *_settleTokens()* as part of his other actions.

Recommended mitigation

Due to the flexibility that is built into the base layer *BookManager*, it can't be fixed there.

It may have to be fixed in *Controller* by applying a reentrancy guard to *Controller.lockAcquired()*. This ensures that one user will not resolve the delta that has been taken on by another user.

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index ece42c4..f0464d5 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -4,6 +4,7 @@ pragma solidity ^0.8.0;
 import {IERC20Permit} from
"@openzeppelin/contracts/token/ERC20/extensions/IERC20Permit.sol";
 import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
 import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
+import {ReentrancyGuard} from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
 import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";

 import "./interfaces/IController.sol";
@@ -11,7 +12,7 @@ import "./interfaces/ILocker.sol";
 import "./interfaces/IBookManager.sol";
 import "./libraries/OrderId.sol";

-contract Controller is IController, ILocker {
+contract Controller is IController, ILocker, ReentrancyGuard {
     using TickLibrary for *;
     using OrderIdLibrary for OrderId;
     using SafeERC20 for IERC20;
@@ -69,7 +70,7 @@ contract Controller is IController, ILocker {
         return tick.toPrice();
     }

-    function lockAcquired(address, bytes memory data) external returns (bytes memory
returnData) {
+    function lockAcquired(address, bytes memory data) external nonReentrant returns
(bytes memory returnData) {
         if (msg.sender != address(_bookManager)) revert InvalidAccess();
         (
             address user,
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

Low severity findings

TRST-L-1 Controller._take() needs to check if the heap is empty

- **Category:** Logical flaws
- **Source:** Controller.sol
- **Status:** Fixed

Description

Controller._spend() will not continue to call *BookManager.take()* when the heap is empty. This is because when the heap is empty, *BookManager.take()* will revert when calling *heap.root()*.

```
    function _spend(SpendOrderParams memory params) internal {
        IBookManager.BookKey memory key = _bookManager.getBookKey(params.id);

        uint256 takenQuoteAmount;
        uint256 leftBaseAmount = params.baseAmount;

        while (leftBaseAmount > 0 && !_bookManager.isEmpty(params.id)) {
```

Similarly, *Controller._take()* should also check whether the heap is empty, which it currently doesn't.

```
    function _take(TakeOrderParams memory params) internal {
        IBookManager.BookKey memory key = _bookManager.getBookKey(params.id);

        uint256 leftQuoteAmount = params.quoteAmount;
        uint256 spendBaseAmount;

        uint256 quoteAmount;
        uint256 baseAmount;
        while (leftQuoteAmount > quoteAmount) {
```

Consider that the quote amount in the book is 1000.

1. Alice and Bob initiate transactions to take 600 **quote** at the same time.
2. Alice's transaction succeeds, but Bob's transaction fails because the heap is empty.

A better approach would be to have Bob take 400 **quote**.

Recommended mitigation

It is recommended to check if the heap is empty in *Controller._take()*.

```
diff --git a/contracts/Controller.sol b/contracts/Controller.sol
index ece42c4..0e53a52 100644
--- a/contracts/Controller.sol
+++ b/contracts/Controller.sol
@@ -212,7 +212,7 @@ contract Controller is IController, ILocker {

         uint256 quoteAmount;
         uint256 baseAmount;
-        while (leftQuoteAmount > quoteAmount) {
+        while (leftQuoteAmount > quoteAmount && !_bookManager.isEmpty(params.id)) {
             unchecked {
                 leftQuoteAmount -= quoteAmount;
                 spendBaseAmount += baseAmount;
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-L-2 Book.calculateClaimableRawAmount() may overflow

- **Category:** Overflow issues
- **Source:** Book.sol
- **Status:** Fixed

Description

Book.calculateClaimableRawAmount() overflows when **orderAmount** is greater or equal to **2^63** and the whole order is claimable.

In this case **orderAmount + totalClaimable > type(uint64).max = 2^64 – 1**.

```
    function calculateClaimableRawAmount(State storage self, Tick tick, uint40 index)
internal view returns (uint64) {
        uint64 orderAmount = self.getOrder(tick, index).pending;

        Queue storage queue = self.queues[tick];
        // @dev Book logic always considers replaced orders as claimable.
        unchecked {
            if (uint256(index) + MAX_ORDER < queue.orders.length) return orderAmount;
        }
        uint64 totalClaimable = self.totalClaimableOf.get(tick);
        uint64 rangeRight = _getClaimRangeRight(queue, index);
        if (rangeRight >= totalClaimable + orderAmount) return 0; // @auidt: may
overflow here

        // -------- totalClaimable ---------|---
        // ------|---- orderAmount ----|--------
        //   rangeLeft           rangeRight
        if (rangeRight <= totalClaimable) return orderAmount;
        // -- totalClaimable --|----------------
        // ------|---- orderAmount ----|--------
        //   rangeLeft           rangeRight
        else return totalClaimable + orderAmount - rangeRight;  // @auidt: may
overflow here
    }
```

The scenario is extremely unlikely since, if the **Book** is configured correctly, one tick needs to have at least the value of half the total supply of the **quote** token.

Still, it should be fixed since the funds could not be recovered.

Recommended mitigation

It is recommended to rearrange the calculations to prevent the overflow.

Note that **rangeRight – orderAmount** can never underflow.

```
diff --git a/contracts/libraries/Book.sol b/contracts/libraries/Book.sol
```

```
index 5a14bc6..a7b7923 100644
--- a/contracts/libraries/Book.sol
+++ b/contracts/libraries/Book.sol
@@ -179,7 +179,7 @@ library Book {
         }
         uint64 totalClaimable = self.totalClaimableOf.get(tick);
         uint64 rangeRight = _getClaimRangeRight(queue, index);
-        if (rangeRight >= totalClaimable + orderAmount) return 0;
+        if (rangeRight - orderAmount >= totalClaimable) return 0;

         // -------- totalClaimable ---------|---
         // ------|---- orderAmount ----|--------
@@ -188,7 +188,7 @@ library Book {
         // -- totalClaimable --|----------------
         // ------|---- orderAmount ----|--------
         //   rangeLeft           rangeRight
-        else return totalClaimable + orderAmount - rangeRight;
+        else return totalClaimable - (rangeRight - orderAmount);
     }
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

In addition, the logic in *Book.calculateClaimableRawAmount()* is now wrapped in an **unchecked** block. This is safe and there is no risk of overflows or underflows.

Additional recommendations

Use latest solmate implementation for ERC20 transfer

The implementation of the ERC20 transfer in *Currency.transfer()* should match the solmate implementation.

There is currently no risk, but the current implementation temporarily overwrites the free memory pointer which should be avoided.

Also, a small fix has been applied to the solmate library to mask the **to** address.

```
diff --git a/contracts/libraries/Currency.sol b/contracts/libraries/Currency.sol
index c8fb4f2..8e81103 100644
--- a/contracts/libraries/Currency.sol
+++ b/contracts/libraries/Currency.sol
@@ -32,27 +32,24 @@ library CurrencyLibrary {
             if (!success) revert NativeTransferFailed();
         } else {
             assembly {
-                // We'll write our calldata to this slot below, but restore it later.
-                let memPointer := mload(0x40)
+                // Get a pointer to some free memory.
+                let freeMemoryPointer := mload(0x40)

                 // Write the abi-encoded calldata into memory, beginning with the
function selector.
-                mstore(0,
0xa9059cbb00000000000000000000000000000000000000000000000000000000)
-                mstore(4, to) // Append the "to" argument.
-                mstore(36, amount) // Append the "amount" argument.
-
-                success :=
-                    and(
-                        // Set success to whether the call reverted, if not we check
it either
-                        // returned exactly 1 (can't just be non-zero data), or had
no return data.
-                        or(and(eq(mload(0), 1), gt(returndatasize(), 31)),
iszero(returndatasize())),
-                        // We use 68 because that's the total length of our calldata
(4 + 32 * 2)
-                        // Counterintuitively, this call() must be positioned after
the or() in the
-                        // surrounding and() because and() evaluates its arguments
from right to left.
-                        call(gas(), currency, 0, 0, 68, 0, 32)
-                    )
-
-                mstore(0x60, 0) // Restore the zero slot to zero.
-                mstore(0x40, memPointer) // Restore the memPointer.
+                mstore(freeMemoryPointer,
0xa9059cbb00000000000000000000000000000000000000000000000000000000)
+                mstore(add(freeMemoryPointer, 4), and(to,
0xffffffffffffffffffffffffffffffffffffffff)) // Append and mask the "to" argument.
+                mstore(add(freeMemoryPointer, 36), amount) // Append the "amount"
argument. Masking not required as it's a full 32 byte type.
+
+                success := and(
+                    // Set success to whether the call reverted, if not we check it
either
+                    // returned exactly 1 (can't just be non-zero data), or had no
return data.
```

```
+                    or(and(eq(mload(0), 1), gt(returndatasize(), 31)),
iszero(returndatasize())),
+                    // We use 68 because the length of our calldata totals up like
so: 4 + 32 * 2.
+                    // We use 0 and 32 to copy up to 32 bytes of return data into the
scratch space.
+                    // Counterintuitively, this call must be positioned second to the
or() call in the
+                    // surrounding and() call or else returndatasize() will be zero
during the computation.
+                    call(gas(), token, 0, freeMemoryPointer, 68, 0, 32)
+                )
            }

            if (!success) revert ERC20TransferFailed();
```

Controller._provider may be out of sync with BookManager.defaultProvider

Controller._provider is set to *BookManager.defaultProvider()* in the constructor of *Controller* and is immutable.

If the owner changes the **defaultProvider** in *BookManager.setDefaultProvider()*, this will make the **Controller._provider** out of sync with **BookManager.defaultProvider**.

The **_provider** is currently not used in the *Controller* so it is recommended to remove this variable.

```
-   address private immutable _provider;

    constructor(address bookManager) {
        _bookManager = IBookManager(bookManager);
-       _provider = _bookManager.defaultProvider();
    }
```

TickLibrary.toPrice() should call validate()

TickLibrary.fromPrice() will call *validatePrice()* to check whether price is within the valid range, while *TickLibrary.toPrice()* does not call *validate()* to check whether tick is within the valid range.

```
    function fromPrice(uint256 price) internal pure validatePrice(price) returns
(Tick) {
...
    function toPrice(Tick tick) internal pure returns (uint256 price) {
```

Both functions are called in the *Controller*'s view function, so the appropriate checks should be applied to both.

```
    function fromPrice(uint256 price) external pure returns (Tick) {
        return price.fromPrice();
    }

    function toPrice(Tick tick) external pure returns (uint256) {
        return tick.toPrice();
    }
```

The recommended fix is to validate the tick:

```
diff --git a/contracts/libraries/Tick.sol b/contracts/libraries/Tick.sol
index 9182965..07e47fe 100644
--- a/contracts/libraries/Tick.sol
+++ b/contracts/libraries/Tick.sol
@@ -76,6 +76,7 @@ library TickLibrary {
     }

     function toPrice(Tick tick) internal pure returns (uint256 price) {
+        validate(tick);
         int24 tickValue = Tick.unwrap(tick);
         uint256 absTick = uint24(tickValue < 0 ? -tickValue : tickValue);
```

Controller should handle ETH and ERC20 balances uniformly

On the one hand, in *_settleTokens()*, the *Controller* does not keep any tokens, it transfers tokens (except ETH) directly from users or to users.

On the other hand, in *_claim()* and *_cancel()*, it allows users to transfer NFTs to the *Controller* and then claim tokens to the *Controller* while these tokens cannot be used in further operations.

This makes the *Controller*'s architecture inconsistent. If users should be allowed to use tokens that are temporarily owned by the *Controller*, then the token accounting needs to be adjusted. If the *Controller* should not keep any tokens, then the NFT transfers should be removed as approvals to the *Controller* are sufficient and less prone to errors.

Improve FeePolicy encoding

The **FeePolicy** is a **uint24** with some of its bits unused (bit 24,23,21)

As a result, it may be checked that all the individual properties of the **BookKey** struct are equal but the book ids may still be different.

In other words, the **FeePolicy** encodes the **useOutput** and **rate** values and there are different **FeePolicy** values that when decoded have the same **useOutput** and **rate** values.

A simple modification of the *FeePolicy* library can get rid of this quirk.

```
diff --git a/contracts/libraries/FeePolicy.sol b/contracts/libraries/FeePolicy.sol
index 86eaf8e..0b6a9c4 100644
--- a/contracts/libraries/FeePolicy.sol
+++ b/contracts/libraries/FeePolicy.sol
@@ -11,7 +11,7 @@ library FeePolicyLibrary {
     int256 internal constant MAX_FEE_RATE = 500000;
     int256 internal constant MIN_FEE_RATE = -500000;

-    uint256 internal constant RATE_MASK = 0x0fffff; // 20 bits
+    uint256 internal constant RATE_MASK = 0x7fffff; // 23 bits

     error InvalidFeePolicy();

@@ -21,7 +21,7 @@ library FeePolicyLibrary {
         }
```

```
         assembly {
-            feePolicy := or(shl(21, useOutput_), add(rate_, MAX_FEE_RATE))
+            feePolicy := or(shl(23, useOutput_), add(rate_, MAX_FEE_RATE))
         }
     }

@@ -33,7 +33,7 @@ library FeePolicyLibrary {

     function useOutput(FeePolicy self) internal pure returns (bool f) {
         assembly {
-            f := shr(21, self)
+            f := shr(23, self)
         }
     }
```

BookManager: remove unused load() functions

The two *load()* functions in *BookManger* are never used, removing them reduces the code size by 0.236KB.

BookManager.cancel(): perform burn before transfer

In *BookManager.claim(),* burn() is called before transfer().

In *BookManager.cancel()*, transfer() is called before burn().

Therefore it's possible to receive a callback in an inconsistent state in *cancel()* since there should not exist an NFT with **pending = 0**.

In addition, this allows to call the hooks in this order which ideally should not be possible:

- *hooks.beforeCancel()*
- *hooks.beforeClaim()*
- *hooks.afterClaim()*
- *hooksafterCancel()*

There is currently no exploitable scenario but we recommend to always perform *burn()* before *transfer()* to reduce attack surface.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..ef133ec 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -223,11 +223,11 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit
{

         if (!makerPolicy.useOutput()) canceledAmount =
_calculateAmountInReverse(canceledAmount, makerPolicy.rate());

+        if (pending == 0) _burn(OrderId.unwrap(params.id));
+
         reservesOf[key.quote] -= canceledAmount;
         key.quote.transfer(owner, canceledAmount);

-        if (pending == 0) _burn(OrderId.unwrap(params.id));
-
```

```
        key.hooks.afterCancel(params, canceled, hookData);

        emit Cancel(params.id, canceled);
```

BookManager should use conservative fee rounding

The rounding in the *BookManager* should always be in the direction such that the protocol rounds up the funds it takes in and rounds down the funds it gives out.

The problem is that in the *BookManager.claim()* function the **provider** fees are currently rounded up instead of down.

Also, due to rounding, **quoteFee** and **baseFee** might become negative so a check should be added such that only positive fees are added to **tokenOwed** such as not to risk a revert due to underflow. Note that effectively increasing **tokenOwed** in this way is safe since any negative **quoteFee** or **baseFee** values are just due to rounding and the protocol doesn't send out more funds than it owns.

To address the rounding, a **reverse** flag has been added in the *BookManager._calculateFee()* function.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 4965f8c..fe95f93 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -157,7 +157,7 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
         int256 quoteDelta = quoteAmount.toInt256();

         if (!params.key.makerPolicy.useOutput()) {
-            quoteDelta -= _calculateFee(quoteAmount, params.key.makerPolicy.rate());
+            quoteDelta -= _calculateFee(quoteAmount, params.key.makerPolicy.rate(),
false);
         }

         _accountDelta(params.key.quote, quoteDelta);
@@ -190,9 +190,9 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
             int256 quoteDelta = quoteAmount.toInt256();
             int256 baseDelta = baseAmount.toInt256();
             if (params.key.takerPolicy.useOutput()) {
-                quoteDelta -= _calculateFee(quoteAmount,
params.key.takerPolicy.rate());
+                quoteDelta -= _calculateFee(quoteAmount,
params.key.takerPolicy.rate(), false);
             } else {
-                baseDelta -= _calculateFee(baseAmount,
params.key.takerPolicy.rate());
+                baseDelta -= _calculateFee(baseAmount, params.key.takerPolicy.rate(),
false);
             }
             _accountDelta(params.key.quote, -quoteDelta);
             _accountDelta(params.key.base, baseDelta);
@@ -264,25 +264,25 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit
{
             FeePolicy makerPolicy = key.makerPolicy;
             FeePolicy takerPolicy = key.takerPolicy;
             if (takerPolicy.useOutput()) {
-                quoteFee = _calculateFee(claimedInQuote, takerPolicy.rate());
+                quoteFee = _calculateFee(claimedInQuote, takerPolicy.rate(), true);
             } else {
```

```
-                 baseFee = _calculateFee(claimableAmount, takerPolicy.rate());
+                 baseFee = _calculateFee(claimableAmount, takerPolicy.rate(), true);
              }
              if (makerPolicy.useOutput()) {
-                 int256 makerFee = _calculateFee(claimableAmount, makerPolicy.rate());
+                 int256 makerFee = _calculateFee(claimableAmount, makerPolicy.rate(),
false);
                  claimableAmount =
                      makerFee > 0 ? claimableAmount - uint256(makerFee) :
claimableAmount + uint256(-makerFee);
                  baseFee += makerFee;
              } else {
-                 quoteFee += _calculateFee(claimedInQuote, makerPolicy.rate());
+                 quoteFee += _calculateFee(claimedInQuote, makerPolicy.rate(), true);
              }
          }

          Book.Order memory order = book.getOrder(tick, orderIndex);
          address provider = order.provider;
          if (provider == address(0)) provider = defaultProvider;
-         tokenOwed[provider][key.quote] += quoteFee.toUint256();
-         tokenOwed[provider][key.base] += baseFee.toUint256();
+         if (quoteFee > 0) tokenOwed[provider][key.quote] += quoteFee.toUint256();
+         if (baseFee > 0) tokenOwed[provider][key.base] += baseFee.toUint256();

          if (order.pending == 0) _burn(OrderId.unwrap(id));

@@ -354,14 +354,14 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit
{
          currencyDelta[locker][currency] = next;
      }

-     function _calculateFee(uint256 amount, int24 rate) internal pure returns (int256)
{
+     function _calculateFee(uint256 amount, int24 rate, bool reverse) internal pure
returns (int256) {
          bool positive = rate > 0;
          uint256 absRate;
          unchecked {
              absRate = uint256(uint24(positive ? rate : -rate));
          }
          // @dev absFee must be less than type(int256).max
-         uint256 absFee = Math.divide(amount * absRate, uint256(_RATE_PRECISION),
positive);
+         uint256 absFee = Math.divide(amount * absRate, uint256(_RATE_PRECISION),
reverse ? !positive: positive);
          return positive ? int256(absFee) : -int256(absFee);
      }
```

Note that in one of the instances in the *BookManager.claim()* function we do not **reverse** the rounding.

Without reversing, the following behavior occurs:

For a positive fee rate, **makerFee** is rounded up -> **claimableAmount** is rounded down.

For a negative fee rate, **makerFee** is rounded up (toward 0) -> **claimableAmount** is rounded down.

In theory, $1\ wei$ less **makerFee** results in $1\ wei$ more **claimableAmount** and vice versa, so the rounding in this instance should not matter.

Since the owner might effectively increase the **makerFee** from negative to 0 later (we can't decrement **tokenOwed**), the more conservative rounding should be on the **claimableAmount** rather than the **makerFee**.

Consider if **claimableAmount** is rounded up and **makerFee** is rounded down. The rounding down in **makerFee** may not be realized since **tokenOwed** is never decremented.

Mitigation Review

It is still recommended to check **quoteFee > 0** and **baseFee > 0** before incrementing **tokenOwed** based on the reasoning provided in the initial report.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 00d6f8c..ef18ed5 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -285,8 +285,8 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
         Book.Order memory order = book.getOrder(tick, orderIndex);
         address provider = order.provider;
         if (provider == address(0)) provider = defaultProvider;
-        tokenOwed[provider][key.quote] += quoteFee.toUint256();
-        tokenOwed[provider][key.base] += baseFee.toUint256();
+        if (quoteFee > 0) tokenOwed[provider][key.quote] += quoteFee.toUint256();
+        if (baseFee > 0) tokenOwed[provider][key.base] += baseFee.toUint256();

         if (order.pending == 0) _burn(OrderId.unwrap(id));
```

Also, the **quoteAmount** assignment in *BookManager.make()* can be rearranged.

```
diff --git a/contracts/BookManager.sol b/contracts/BookManager.sol
index 00d6f8c..b2dcfd1 100644
--- a/contracts/BookManager.sol
+++ b/contracts/BookManager.sol
@@ -161,8 +161,8 @@ contract BookManager is IBookManager, Ownable2Step, ERC721Permit {
             quoteDelta = int256(quoteAmount);
             if (params.key.makerPolicy.usesQuote()) {
                 quoteDelta += params.key.makerPolicy.calculateFee(quoteAmount,
false);
+                quoteAmount = uint256(quoteDelta);
             }
-            quoteAmount = uint256(quoteDelta);
         }

         _accountDelta(params.key.quote, quoteDelta);
```

Final Mitigation Review

The fix implements the recommendation.

Centralization risks

Hooks are fully trusted

Each **Book** can be configured to use a **Hook** that receives a callback when certain actions are executed. The **Hook** is privileged to perform actions on behalf of the user interacting with the *BookManager*. In particular, the **Hook** can call functions that are protected with the **onlyByLocker** modifier and must not revert as this can lock funds forever. As such, a **Hook** is fully trusted in the context of the **Book** that it is set for.

BookManager owner can set whitelisted providers

In the *BookManager*, only **providers** can receive trading fees and they must be whitelisted by the **owner** of the *BookManager*. Moreover, once a **provider** has been whitelisted, it can be delisted at any time, being unable to earn fees in the future.

The **owner** of the *BookManager* does not have any privileges beyond this and there exist no other trusted roles in the base layer DEX.

.

Systemic risks

Books inherit risks of external tokens

Each **Book** has a **quote** and a **base** currency. Currencies can be native ETH or ERC20 tokens. The **Book** inherits the risks of its underlying tokens. These risks include, but are not limited to, centralization risks, risks of vulnerabilities being exploited and depeg risks. In other words, a **Book** is only as secure as its underlying tokens.

The tokens also need to be compatible with the protocol. For example, fee-on-transfer tokens or rebasing tokens are not supported.

Risks in the underlying tokens of one **Book,** however, do not translate to other **Books** that use different tokens since from a security perspective they are isolated from each other.

Books can be griefed with low liquidity ticks

Griefers can call *BookManager.make()* to create orders at ticks below market price such that any taker first needs to take these orders before reaching legitimate liquidity. By creating each of n orders on a new tick, the taker needs to call n times *BookManager.take()* and incur n times the gas fee for calling the function.

Let's formalize the issue to get a better understanding for why it is a systemic risk and needs to be considered on a per **Book** basis.

For the sake of simplicity, it is assumed that the defined variables are the same for all n orders. In reality we have an average over the sum of the individual orders. So, the variables can be seen as this average over the sum.

Let g_m be the cost in USD of calling *BookManager.make()* once.

Let p be the price in USD at which $1\ wei$ of **quote** is offered.

Let v be the value of $1\ wei$ of **quote** in USD

Let u be the amount of wei per unit of **quote**.

Let n be the number of units that are used to perform the griefing attack. There is one order per unit.

Let g_t be the cost in USD of calling *BookManager.take()* once (this is generally less than calling *BookManager.make()* based on the gas benchmarks in the test suite).

Creating the grief thus costs:

$$cost_{grief} = (g_m + (v - p) * u) * n$$

And resolving the grief costs:

$$cost_{ungrief} = (g_t + (p - v) * u) * n$$

If it is possible to make the cost of ungriefing negative by choosing **Book** parameters, the attack can be mitigated without further economic considerations.

We set:

$$cost_{ungrief} < 0$$

$$\leftrightarrow (g_t + (p - v) * u) * n < 0$$

We know that n is greater zero:

$$\leftrightarrow g_t + (p - v) * u < 0$$

$$\leftrightarrow g_t < (v - p) * u$$

We can see that $cost_{ungrief}$ is negative when g_t is less than the difference between the market value v for $1\ wei$ of **quote** and the offered price p for $1\ wei$ of **quote** times the unit size u.

This assumes that the difference $v - p$ can be realized. E.g., the ungriefer already holds **quote** and so the marginal cost of trading is zero.

It's not possible to prevent this attack be setting **Book** parameters as the griefer can set p close to v such that for any reasonable value of u, $(v - p) * u$ is smaller than g_t.

If the cost of resolving the grief is positive, we can observe the following behavior from rational market participants:

1. Rational takers will only resolve the grief if they can't get a better deal from other exchanges, requiring makers to offer at sub-market prices.
2. Rational makers won't offer at sub-market prices.
3. Market forces will dry up the **Book** and it will be unused.

Essentially this acts as a spread that can be freely determined by the griefer and the lower the liquidity the easier the **Book** is to grief.

Let us now characterize the problem from the griefer's perspective.

The griefer wants to lose as little funds as possible while maximizing the cost for the ungriefer. Naively, this is achieved by setting p one tick below v. However, this is not possible since it risks the grief being resolved by natural market fluctuations.

So, the griefer needs to set p at a distance from v such that market fluctuations don't reach it and the grief needs to be resolved by takers taking a loss.

Choosing the best griefing strategy thus becomes a difficult optimization problem that requires assumptions about takers and their willingness to take orders at prices below market value. It also requires considerations regarding expected market volatility. And finally, it needs to be determined how much the griefer is even willing to pay since there is no financial reward for him. Trying to answer these questions is beyond the scope of the smart contract audit.

NFT transfers can be front-run with claim() and cancel()

When order NFTs are traded on secondary markets or sent from user to user, there is a risk that *claim()* or *cancel()* is called before the transfer to make the NFT worthless.

This is particularly a problem for *claim()* since anyone can call it even if the NFT is stored in an escrow and so the escrow needs to be constructed to be able to process the claimed funds.

Receiving a raw transfer of an order NFT is thus unsafe and can result in the complete loss of funds.