OpenZeppelin | security

OpenZeppelin Uniswap Hooks v1.1.0 RC 1 Audit

UNISWAP FOUNDATION

July 3, 2025

Table of Contents

Table of Contents 2

Summary 3

Scope 4

System Overview 5

AntiSandwichHook 5

LiquidityPenaltyHook 5

LimitOrderHook 6

Differences from the Previous Version 6

Critical Severity 8

C-01 JIT Liquidity Penalty Can Be Bypassed 8

C-02 Asymmetric First In-Block Swap Initialization Leads to Stale State 9

High Severity 10

H-01 Limit Orders Can Be Incorrectly Filled 10

H-02 Accrued Limit Order Fees Can Be Stolen 12

H-03 Liquidity Penalty Can Be Circumvented Using Secondary Accounts 13

H-04 Incorrect Unspecified Amount Handling Breaks Anti-Sandwich Logic 14

H-05 Sandwich Attack Possible via JIT Attack in AntiSandwichHook 15

H-06 Incorrect Handling of unspecifiedAmount Leads to Overcharging 16

H-07 Integer Underflow in _getTargetOutput Due to Improper Type Casting 17

H-08 Missing Liquidity Total Update in withdraw 17

H-09 Incorrect Timing of removingAllLiquidity Flag Calculation in cancelOrder 19

Medium Severity 19

M-01 Block Number Dependency 19

M-02 Incorrect Usage of Returned Balance Delta 20

M-03 Anti-Sandwich Mechanism Also Breaks Price Reliability at the Start of a Block 21

M-04 Incorrect slot0 Persistence on !zeroForOne Swaps Causes Inconsistent Pricing Logic 21

Low Severity 22

L-01 Incomplete Docstrings 22

Notes & Additional Information 23

N-01 Missing Named Parameters in Mappings 23

N-02 Custom Errors in require Statements 23

N-03 Unused Error 24

Conclusion 25

Summary

Type	DeFi
Timeline	From 2025-04-25 To 2025-05-12
Languages	Solidity

Total Issues	19 (0 resolved)
Critical Severity Issues	2 (0 resolved)
High Severity Issues	9 (0 resolved)
Medium Severity Issues	4 (0 resolved)
Low Severity Issues	1 (0 resolved)
Notes & Additional Information	3 (0 resolved)

Scope

This audit report details the comprehensive analysis performed on a set of custom Uniswap V4 hooks. These hooks have been specifically designed to enhance the functionality and security of Uniswap V4 liquidity pools.

The audit was performed on the release-v1.1.0-rc.1 branch of the OpenZeppelin/uniswap-hooks repository at commit 0879747.

In scope were the following files:

```
src
├── base
│   ├── BaseAsyncSwap.sol
│   ├── BaseCustomAccounting.sol
│   ├── BaseCustomCurve.sol
│   └── BaseHook.sol
├── fee
│   └── BaseDynamicAfterFee.sol
├── general
│   ├── AntiSandwichHook.sol
│   ├── LimitOrderHook.sol
│   └── LiquidityPenaltyHook.sol
└── interfaces
    └── IHookEvents.sol
└── utils
    └── CurrencySettler.sol
```

Outside the `general` directory, for which a full line-by-line audit has been performed, the rest of the scope has been audited on its diff against commit cb6d90c.

System Overview

`AntiSandwichHook`

The `AntiSandwichHook` contract implements a sandwich-resistant Automated Market Maker (AMM) design intended to mitigate sandwich attacks, whereby malicious actors exploit the transaction order within a block to extract value at the expense of ordinary users. This is achieved by enforcing that no trade is executed at a more favorable price than what was available at the start of the current block.

At the beginning of each block, the hook records a checkpoint of the pool's price and state. When the first swap of the block occurs, this checkpoint is saved and used as a baseline. Subsequent swaps in the same block are then compared against this checkpoint. For certain trade directions, the hook restricts users from receiving a better execution price than what was available at the block's start, effectively neutralizing the typical profit opportunities exploited in sandwich strategies.

To enforce this, the hook uses a mechanism where excess output tokens, when a swap would otherwise execute at a better price, are withheld and donated back to the pool, benefiting all liquidity providers. This creates a fairer execution environment and significantly reduces the economic incentives for front-running and back-running attacks.

`LiquidityPenaltyHook`

The `LiquidityPenaltyHook` contract is designed to protect against Just-In-Time (JIT) liquidity attacks, whereby sophisticated actors rapidly add and remove liquidity around large trades to capture fees without taking on meaningful market risk. This form of behavior can erode the returns of long-term liquidity providers and reduce the predictability and fairness of fee distribution.

To mitigate this, the hook introduces a time-based penalty mechanism. Specifically, it tracks the block number when liquidity is added and checks how many blocks have passed when liquidity is removed. If liquidity is removed too soon, within a configurable block offset threshold, the hook penalizes the position by donating a portion (up to 100%) of the earned fees back to the pool. The penalty is linearly scaled depending on how soon the removal occurs relative to the configured threshold.

`LimitOrderHook`

The `LimitOrderHook` contract is a hook meant to allow users to create limit orders leveraging Uniswap v4 out-of-range liquidity positions. When a user creates an out-of-range liquidity position with a tick range width of 1 `tickSpacing`, only one of the two assets is required, effectively simulating the process of placing an order into an order book at a specific price (the tick).

When swapping makes it so that the tick crosses the lower and upper tick of the *order* tick range, the order can be considered filled, since the swap effectively consumed the liquidity present in that tick range and swapped it for the opposite asset. The contract also features additional functionalities such as the following:

- Users' orders are cumulated if they are within the same tick ranges, which is exactly how an order book would do it.
- Users can cancel their orders using the `cancelOrder` function if they have not been filled. If an order is canceled, the fees that such liquidity position might have generated are distributed pro bono to all members of the same liquidity position (the same order in the order book) or withdrawn by the user if theirs is the last liquidity in that specific range.
- Users can withdraw the proceeds of their filled orders through the `withdraw` function.

Differences from the Previous Version

Apart from the new contracts, some changes have also been made to the existing contracts:

- The `IHooksEvents` interface has been added, which defines some common event emissions. In addition, the `BaseAsyncSwap`, `BaseCustomAccounting`, `BaseCustomCurve`, and `BaseDynamicAfterFee` contracts have been modified to emit the corresponding events where appropriate.
- The `CurrencySettler` library has been modified to include the usage of `SafeERC20`, and to return early when amounts are 0 since some tokens might revert with such values.
- The `BaseAsyncSwap` contract now has an `internal` and `override`able `_calculateSwapFee` function that can return the amount of swap fees to apply in case it is needed (currently returning 0).
- The `BaseCustomAccounting` contract now supports the usage of `salt`s for liquidity positions, so that users can mark their unique positions by providing a `salt` value. It also features a new `_handleAccruedFees` function to handle fees accrued in liquidity positions.

- Similarly, the `BaseCustomCurve` contract now has an `override`able `_getSwapFeeAmount` function to calculate the fees collected by a swap.

Critical Severity

C-01 JIT Liquidity Penalty Can Be Bypassed

The `LiquidityPenaltyHook` contract implements a mechanism to penalize JIT liquidity provision by imposing a penalty fee on positions that are added and removed within a short timeframe (defined by `blockNumberOffset`). This aims to prevent sandwich attacks where traders add liquidity just before a large swap, collect fees from that swap, and then immediately withdraw their liquidity.

When liquidity is removed before the configured `blockNumberOffset` number of blocks has passed, the hook calculates a penalty based on the fees earned (`feeDelta`) and donates this penalty back to the pool, effectively reducing the profit from JIT liquidity tactics. However, there is a vulnerability that allows a complete bypass of the penalty mechanism through a sequence of operations leveraging Uniswap v4's fee collection behavior. In Uniswap v4, when an `increaseLiquidity` operation is performed on an existing position, it automatically collects all accrued fees and credits them to the user, resetting `feesOwed` to zero.

The `LiquidityPenaltyHook` calculates penalties based on `feeDelta` from the `afterRemoveLiquidity` hook, which represents uncollected fees at the time of liquidity removal. By strategically performing an `increaseLiquidity` operation with a minimal amount before removing all liquidity, an attacker can collect all fees separately from the removal action, resulting in a `feeDelta` of zero during the liquidity removal and, thus, avoiding the intended penalty entirely.

1. Attacker adds substantial liquidity position to a pool.
2. Target swap executes, generating fees for the attacker's position.
3. Attacker calls `increaseLiquidity` with a minimal amount (e.g., 1 wei), which:
    - collects all accrued fees and transfers them to the attacker
    - calls `_afterAddLiquidity` hook (which only records the block number but does not penalize)
    - resets `feesOwed` to zero
4. Attacker immediately removes all liquidity. This calls `_afterRemoveLiquidity`, but since `feeDelta` is now zero, the penalty calculation yields zero.

This vulnerability completely undermines the core security mechanism of the `LiquidityPenaltyHook` contract. It allows JIT liquidity providers to execute sandwich

attacks with zero penalty, defeating the entire purpose of the contract. Since the penalty can be fully bypassed, there is effectively no protection against JIT manipulation, leaving pools vulnerable to the exact attacks that this hook was designed to prevent.

The economic impact is significant since attackers can extract value from ordinary traders through JIT tactics without incurring the intended penalties, resulting in a direct transfer of value from normal users to manipulators.

The simplest immediate fix would be to extend the hook permissions to also monitor `beforeAddLiquidity` events and track fee collection that occurs during liquidity increases, accounting for these collected fees when calculating penalties during eventual liquidity removal.

C-02 Asymmetric First In-Block Swap Initialization Leads to Stale State

The `AntiSandwichHook` contract aims to prevent sandwich attacks by ensuring that swaps adhere to prices from the beginning of the block. It uses a checkpoint, stored in the `_lastCheckpoints` mapping, to calculate an expected output in `_getTargetOutput`.

The primary issue is how the `_lastCheckpoints[poolId].state` (specifically its `ticks` component) is populated, leading to potentially stale or incomplete data.

1. In `_beforeSwap`, for a new block, only `_lastCheckpoints[poolId].slot0` (containing the initial tick and `sqrtPriceX96`) is updated from the pool's current state.
2. The more comprehensive `_lastCheckpoints[poolId].state` (which includes `state.ticks`, `state.liquidity`, and its own `state.slot0`) is intended to be initialized or updated in the `_afterSwap` function, specifically after the first swap of the block.
3. The logic in `_afterSwap` to populate `_lastCheckpoints[poolId].state.ticks` iterates using `for (int24 tick = _lastCheckpoint.slot0.tick(); tick < tickAfter; tick += key.tickSpacing)`. Here, `_lastCheckpoint.slot0.tick()` refers to the tick at the start of the block (captured in `_beforeSwap`), while `tickAfter` is the tick *after* the first swap has been completed.

The problem arises if the first swap in a block does not result in `tickAfter` being strictly greater than `_lastCheckpoint.slot0.tick()` (e.g., the tick remains the same or decreases). In this scenario, the `for` loop in line 123 will not execute. Consequently,

`_lastCheckpoints[poolId].state.ticks` will not be populated with fresh data for the current block. It will retain stale tick data from a previous block or remain uninitialized.

Critically, even if this loop does not execute, `_lastCheckpoint.blockNumber` *is* updated to the current `block.number` in line 119. This prevents any subsequent calls to `_afterSwap` within the same block from re-attempting to initialize these tick values, as the `_lastCheckpoint.blockNumber != blockNumber` condition in line 118 will be `false`.

If the first swap in a block does not advance the tick (or moves it backward), `_lastCheckpoint.state.ticks` is not updated for that block. Subsequent swaps in the same block will then call `_getTargetOutput`, which relies on this `_lastCheckpoint.state` for its simulation. The simulation will use stale or uninitialized tick data, leading to an incorrect `targetOutput` and potentially negating the anti-sandwich protection.

Consider initializing the full start-of-block snapshot, including all necessary components of `_lastCheckpoints[poolId].state` (such as `slot0`, `liquidity`, and a relevant range of `ticks`), within the `_beforeSwap` hook when a new block is detected. This approach would ensure that the checkpoint is based purely on the state at the beginning of the block, independent of the effects or tick movements of the first swap. This would involve fetching tick data around `_lastCheckpoint.slot0.tick()` at the beginning of the block.

High Severity

H-01 Limit Orders Can Be Incorrectly Filled

The `LimitOrderHook` contract uses the `_getCrossedTicks` function to determine which ticks were crossed during a swap by comparing the current pool tick with the previously stored tick. It then processes any limit orders within that range as filled in the `_afterSwap` function.

However, there is an edge case where limit orders can be incorrectly filled when one of the following happens:

- A swap moves the tick into a limit order range (crossing the lower boundary but not the upper boundary).
- A subsequent swap moves the tick back in the opposite direction (crossing the lower boundary again).

This is demonstrated by the following scenario:

Consider a pool with:

```
Current tick: -200
Tick spacing: 10
Limit order in range: [0, 10]
```

Sequence of events:

1. A swap moves the tick to 5 (increasing tick, crosses the lower boundary at 0). `tickLowerLast` is set to 0 (the highest tick that was crossed).
2. A swap then moves the tick from 5 to -5 (decreasing tick, crosses the lower boundary again).
3. `_getCrossedTicks` calculates:

```
tickLower = -10 (current tick rounded down)
tickLowerLast = 0 (from previous swap)
```

and it returns (-10, 0, 0) as the range of crossed ticks.

4. The for loop in `_afterSwap` iterates through this range exactly once, hitting tick 0. The limit order at [0, 10] is marked as filled and deleted from the pool.

This is incorrect because the limit order was only partially filled (the price never crossed the upper boundary at tick 10).

The limit order filling mechanism should be modified to properly track when an order is fully filled by ensuring both boundaries of the order range have been crossed. One of the following approaches could be taken to address this:

- Track both the lower and upper boundaries of limit orders
- Only mark an order as filled when both boundaries have been crossed
- Alternatively, implement a mechanism to handle partial fills, allowing users to claim the filled portion while keeping the remainder of the order active.

At minimum, consider revising the `_getCrossedTicks` function to properly handle the case when a tick is crossed multiple times in opposite directions to avoid prematurely marking orders as filled.

H-02 Accrued Limit Order Fees Can Be Stolen

In the `LimitOrderHook` contract, a limit order is one `tickSpacing` wide and is considered *filled* when the tick goes from the lower tick of the `tickSpacing` to the upper tick.

In Uniswap, a tick is a small price difference, but the `tickSpacing` is multiple ticks. For example, the default tick spacing in a 1% fee pool is 200, which corresponds to a price difference between `tickSpacings` of 2.02%. With this significant price range between `tickSpacing`, the price can land within a limit order and multiple swaps can occur without fully crossing over to the other side of `tickSpacing`, which would lead to significant accrual of fees for the liquidity that makes up the limit order.

When limit orders are filled, the fees are added to the tokens that are claimable as part of the `delta`. Upon withdrawal, the total token amount is divided based on how much liquidity a user has provided.

Suppose that an unfilled limit order has accrued significant fees. The attacker can execute this sequence in one block:

1. Create a large limit order, capturing 90% of the limit order liquidity for that tick spacing.
2. Swap so that the `tickSpacing` is crossed and the limit order is filled.
3. Withdraw once filled the limit order.

Since all the fees collected by the limit order liquidity are split based on the share of liquidity when the order is filled, regardless of how much of the fees they were *responsible for* or how long their position was active, the attacker captures the majority of fees despite providing liquidity for less than 1 block. They capture fees that were accrued before they were ever an active liquidity provider. Moreover, the comment on canceled orders mentions that the fee is given to the existing order. It does not take into account that future liquidity providers can claim/steal a portion of these fees.

Consider tracking the timestamp or block number when each user adds liquidity, and distributing fees only based on the portion of fees accrued after that point. This would prevent newly added liquidity, especially JIT additions, from unfairly claiming fees that were accumulated before their participation. Alternatively, maintaining per-user fee snapshots would allow precise and fair distribution of rewards.

H-03 Liquidity Penalty Can Be Circumvented Using Secondary Accounts

The `LiquidityPenaltyHook` contract is designed to penalize JIT liquidity provision by donating fees to the pool when liquidity is added and removed within a short time window (defined by `blockNumberOffset`). This discourages attackers from earning fees unfairly from the swaps they anticipate. However, the penalty mechanism can be bypassed using a coordinated attack involving two accounts, exploiting how fee donation rewards in-range liquidity providers at the time of removal.

Below is an example simulating how an attacker could bypass the JIT protection using two coordinated accounts:

1. **Setup:** Current tick is in a stable range (e.g., 10,000).
2. **Step 1 – Positioning:** Attacker's secondary account (Account B or "B") adds a small liquidity position in an out-of-range tick window (e.g., [-13,000, -12,000]) where no other liquidity exists.
3. **Step 2 – Fee Capture:** In the next block:
    - Primary account (Account A or "A") adds large liquidity in the current tick.
    - Victim executes a large swap, generating fees for A.
4. **Step 3 – Fee Redirection:**
    - A swaps to move the price into B's tick range.
    - A removes liquidity. Since removal happens within the offset window, `LiquidityPenaltyHook` forcibly donates accrued fees back to the pool.
    - B, now the sole in-range provider, removes liquidity and receives the donated fees.
    - A swaps again to restore the original tick range.

This sequence allows Account B to collect fees originally earned by Account A, effectively bypassing the penalty that would otherwise reduce A's profits. Since the hook donates to whoever is in range during removal, and has no mechanism to detect coordination between A and B, this type of attack is feasible and profitable.

Consider enhancing the hook logic to monitor for abrupt and large tick changes, particularly those occurring within a single block. If such behavior is detected, fee donations could be withheld, delayed, or proportionally distributed over time to prevent their full capture by opportunistically positioned accounts. In addition, donation eligibility could be conditioned on the continuity and persistence of the beneficiary's liquidity, reducing the incentive for short-lived, strategically placed positions that aim to exploit transient price shifts.

H-04 Incorrect Unspecified Amount Handling Breaks Anti-Sandwich Logic

The current implementation of the `AntiSandwichHook` contract assumes that the "unspecified amount" in a swap always refers to the output amount the user will receive. However, it can also represent the input amount that the user is willing to pay. Due to this flawed assumption, the anti-sandwich protection becomes ineffective in certain conditions, allowing an attacker to successfully execute a sandwich attack.

Exploit Scenario

1. Attacker initiates a swap from $token_0$ to 10,000 $token_1$ (`zeroForOne = true`).
    - `unspecified amount`: the amount (input) the attacker must pay: 10,000
    - `targetOutput`: set to 10,000, as that is what the attacker receives.
2. Victim follows up with a swap from 10,000 $token_0$ to 9,500 $token_1$ (`zeroForOne = true`).
3. Attacker performs the closing leg of the sandwich, swapping 10,000 $token_1$ back to $token_0$ (`zeroForOne = false`).
    - The attacker specifies only how much they want to receive, e.g., 12,000 $token_0$.
    - `unspecified amount`: now represents how much $token_1$ the attacker is willing to pay, say 10,000.
    - The code calculates `targetOutput` using the initial state from the first trade (before the victim's), which gives a value of 12,000.
    - Since `targetOutput > unspecified amount`, the hook incorrectly **caps `targetOutput` to 10,000**, which does not trigger a fee or block the swap.

As a result, the attacker successfully extracts profit — swapping 10,000 $token_1$ for 12,000 $token_0$ — completing the sandwich attack.

A similar attack is also possible in the reverse direction, such as: `!zeroForOne → !zeroForOne → zeroForOne`.

The vulnerability stems from treating the "unspecified amount" exclusively as the amount to be received instead of handling both input and output cases correctly.

To prevent this class of sandwich attacks, the anti-sandwich hook must distinguish whether the unspecified amount represents input or output. In case it represents the amount to be paid (i.e., input), then:

- if the unspecified amount is greater than `targetOutput`, then `targetOutput` must be set equal to the unspecified amount

- if the `targetOutput` is greater than the unspecified amount, then the user must end up paying the `targetOutput` amount instead

H-05 Sandwich Attack Possible via JIT Attack in `AntiSandwichHook`

The `AntiSandwichHook` contract attempts to mitigate sandwich attacks by ensuring that swaps do not execute at prices better than those available at the beginning of the block. This is enforced through an additional fee charged on the swap, which is then donated to the pool and distributed to active liquidity providers (LPs) at the tick where the swap concludes.

However, a profitable sandwich strategy remains viable due to the ability of attackers to manipulate liquidity positions within the same block. The vulnerability emerges from the fact that liquidity provision and removal are permissionless and costless (excluding gas), and the redistribution of the penalized amount is proportional to the LP share at the final tick of the swap.

A possible attack scenario is as follows:

1. **Alice initiates a `!zeroForOne` swap**, setting the price checkpoint for the block.
2. **Bob (victim) performs a second `!zeroForOne` swap**, receiving a worse price.
3. **Alice adds a large, concentrated liquidity position** in the tick where the pool price will land, acquiring ~99% of the active liquidity.
4. **Alice executes a `zeroForOne` swap**, triggering `_afterSwapHandler`, which donates the fee to the pool.
5. **Alice receives 99% of the penalty** back via her dominant liquidity share.
6. **Alice withdraws the liquidity**, capturing a profit despite the intended penalty.

Consider implementing a protection mechanism against JIT liquidity attacks to prevent users from briefly injecting large amounts of liquidity immediately before fee redistribution events.

H-06 Incorrect Handling of `unspecifiedAmount` Leads to Overcharging

The dynamic-fee hooks (`BaseDynamicAfterFee` → `AntiSandwichHook`) assume that the `unspecifiedAmount` in a swap is always the output that the user will receive. However, it can also be the input that the user must pay.

- **When `unspecifiedAmount` is output**: capping it by `_targetOutput` makes sense, as it prevents the user from receiving more than the fair amount.
- **When `unspecifiedAmount` is input**: capping by `_targetOutput` is incorrect. If the user's intended input exceeds `_targetOutput`, the code silently treats the difference as a "fee," minting extra tokens and charging the user more than intended.

Illustrative scenario

1. First swap in the block: user wants to receive 10,000 token_0
   - implies an unspecified input of 9,000 token_1
2. Second swap in the same block: user again wants 10 000 token_0, but now must pay 15 000 token_1.
   - `_targetOutput` will be `9,000`, because of the initial state of the block.
   - `unspecifiedAmount` will be `15,000`
   - It calculates `fee = unspecifiedAmount − _targetOutput = 15,000 − 9,000 = 6,000`, mints 6,000 extra token_1, and effectively charges the user 21,000 instead of 15,000.

Consider adjusting the cap logic in `_afterSwap` to distinguish input vs. output cases. When the `unspecifiedAmount` represents the user's input (the amount they pay), then one of the following must be done:

- If the unspecified amount is greater than `targetOutput`, then `targetOutput` must be set equal to the unspecified amount.
- If `targetOutput` is greater than the unspecified amount, then the amount the user pays must be equal to `targetOutput`.

This way, the fee will not be applied since `targetOutput == unspecifiedAmount`, and it can always be ensured the user pays the highest amount.

H-07 Integer Underflow in `_getTargetOutput` Due to Improper Type Casting

In the `AntiSandwichHook` contract, an integer underflow may occur due to the improper casting of a negative `int128` value to `uint128` in the `_getTargetOutput` function:

```
int128 target =
    (params.amountSpecified < 0 == params.zeroForOne) ? targetDelta.amount1() :
targetDelta.amount0();
targetOutput = uint256(uint128(target));
```

To understand the issue, consider the following example:

```
SwapParams({
    zeroForOne: true,
    amountSpecified: 10_000 * 1e18,
    sqrtPriceLimitX96: sqrtPriceLimitX96
});
```

In this scenario, the user specifies they want to *receive* 10,000 units of token1. Since `amountSpecified` is positive and `zeroForOne` is `true`, the swap direction is token0 → token1, and the *unspecified amount* — the amount of token0 that the user must pay — will be negative.

The logic in `_getTargetOutput` selects this negative `int128` value (`target`), directly casts it to `uint128`, and then to `uint256`, without correcting the sign. This results in an underflow, setting `targetOutput` to a very large, incorrect value (e.g., `2**128 - 1`).

An attacker could exploit this vulnerability to carry out a profitable sandwich attack. Specifically, they could execute the final swap in a block with a positive `amountSpecified`, which causes the `target` (the unspecified input amount) to be negative. As a result, the anti-sandwich mechanism would interpret this as an extremely high `targetOutput`, allowing the attacker to extract the expected profits without penalties, completing the sandwich strategy successfully.

Consider performing a sign check and normalizing before casting to unsigned integers.

H-08 Missing Liquidity Total Update in `withdraw`

The `LimitOrderHook` contract contains an issue in its `withdraw` function where the total liquidity counter is not properly updated when users withdraw their liquidity from filled orders. When a user withdraws their funds, the contract correctly deletes the individual user's liquidity

entry with `delete orderInfo.liquidity[msg.sender]`, but fails to decrease `orderInfo.liquidityTotal` accordingly.

This oversight means that the `liquidityTotal` variable continues to reflect an inflated value that includes withdrawn liquidity. Since this value is used as the denominator when calculating proportional token distribution for subsequent withdrawals (`amount0 = FullMath.mulDiv(orderInfo.currency0Total, liquidity, liquidityTotal)`), users withdrawing later will receive systematically smaller amounts than they should. This is mainly because `currencyXTotal` is also decreased at each withdrawal, losing the percentage ratio of owned shares within the order.

The impact of this issue increases with each withdrawal. As more users withdraw their liquidity, the discrepancy between the stored total liquidity and the actual remaining liquidity grows larger. In the worst-case scenario, if a significant portion of users have withdrawn, but the total has not been updated, the final users might receive substantially less than their fair share of tokens. In addition, if all users eventually withdraw, some tokens will remain locked in the contract due to division calculations using an artificially high denominator.

An example is as follows:

- Two users have both deposited 500 of liquidity, so `totalLiquidity` is 1000.
- The order is filled and there's now 1000 of `currency0`.

The expected outcome is that both users will receive 500 of `currency0` but let us see what happens: when the first user withdraws, `amount0` is `1000 * 500 / 1000` which is 500. After this withdrawal, `currency0Total` is 500 and `totalLiquidity` is 1000. If the second user withdraws at this point, `amount0` will be calculated as `500 * 500 / 1000` which ends up being 250 and not 500.

To address this issue, the `withdraw` function should be modified to update the total liquidity counter whenever a user withdraws. A simple subtraction could be added after the individual liquidity entry is deleted: `orderInfo.liquidityTotal -= liquidity;`.

This small change would ensure that the total liquidity accurately reflects the remaining liquidity in the order at all times, maintaining proportional withdrawals for all participants.

H-09 Incorrect Timing of `removingAllLiquidity` Flag Calculation in `cancelOrder`

The `LimitOrderHook` contract contains a logical flaw in its `cancelOrder` function pertaining to the calculation of the `removingAllLiquidity` flag. This flag determines how fees are allocated in the `_handleCancelCallback` function, but it is being computed at the wrong moment in the execution sequence.

In the current implementation, the contract first decrements the user's liquidity from `orderInfo.liquidityTotal` and then compares the user's liquidity amount with the updated total. Since the total has already been reduced by the user's contribution, this comparison will always be evaluated to `false`. This means that even when a user cancels the last remaining liquidity in an order, the `removingAllLiquidity` flag is incorrectly set to `false`.

This issue impacts fee distribution, as explained in the `_handleCancelCallback` function. When a user cancels the last of an order's liquidity, the fees should go to them instead of being allocated to non-existent remaining liquidity providers. However, the current implementation incorrectly holds these fees in the contract where they become apparently inaccessible. There is also a second issue that allows these fees to be stolen: at the order key in the `orders` mapping, the ID is not being reset to `ORDER_ID_DEFAULT`. This means that someone can place an order, fill it, and place it again using the same order key as before. This will cause the order to be effectively fillable again, and the stuck fees would become available to be taken.

To fix this issue, the `removingAllLiquidity` check should be performed before reducing the total liquidity, and the `cancelOrder` or its callback should also reset the `orderId` inside the `orders` mapping. These changes ensure that when the last user cancels their liquidity, they correctly receive any accrued fees, maintaining proper accounting within the contract and preventing funds from being locked indefinitely.

Medium Severity

M-01 Block Number Dependency

The `AntiSandwhichHook` and `LiquidityPenaltyHook` hooks both rely on `block.number` to establish whether the price should be taken as the one at the beginning of

the block in the former and whether the liquidity provider should receive a penalty for liquidity provisioning on a very short timeframe in the latter.

Generally, relying on `block.number` is a good decision. However, some blockchains, especially L2s like Arbitrum, will return the L1 block number within the EVM. This means that on chains like Arbitrum, the `block.number` constraints enforced by both hooks can extend for several L2 blocks, since many of them will reflect the same L1 block when accessed through `block.number`.

This means that the fixed price of the `AntiSandwichHook` and the liquidity penalty for the `LiquidityPenaltyHook` will last for several L2 blocks instead even when meant to last for just 1 block. In such cases, the hooks would be unexpectedly over-constrained. In the specific case of Arbitrum, one can access the L2 block by using `ArbSys(100).arbBlockNumber()`, but, in general, one could also think about using `block.timestamp` instead.

Consider clarifying, both from a design perspective and at the implementation level, how the contracts should behave in such cases.

M-02 Incorrect Usage of Returned Balance Delta

In the `_handlePlaceCallback` function of the `LimitOrderHook` contract, the contract uses the raw `delta` values returned from `modifyLiquidity` to determine whether the limit order is correctly placed out of range. The issue is that the returned delta represents the sum of principal amount and any accrued fees from the position. Since accrued fees are non-negative, they can potentially change the sign of the delta value from what would be expected based on the principal alone.

When placing an order in a tick that has accumulated fees, the principal component would be negative (providing liquidity), but the accumulated fees could be positive. If the accumulated fees exceed the principal amount, the sign of the total delta could flip. This would trigger a revert when trying to convert the positive delta into a negative value, before casting to a `uint`.

Consider modifying the `_handlePlaceCallback` function to properly account for fees in the balance delta. The most reliable approach is to get the principal by subtracting the fee component from the value returned by the pool manager.

M-03 Anti-Sandwich Mechanism Also Breaks Price Reliability at the Start of a Block

`AntiSandwichHook` tries to stop sandwich attacks by making sure that swaps cannot happen at a better price than what was available at the start of the block. This only works if that starting price is fair. In regular Uniswap pools, it is hard to manipulate the price at the start of a block because doing so would leave the pool exposed to arbitrage. So, normally, the price at the start of a block is close to the real market price.

However, this assumption becomes invalid once the anti-sandwich mechanism is applied. Since the hook makes any within-block price improvements non-arbitrageable (by redistributing the excess back to LPs), external arbitrageurs are disincentivized from restoring the price to its fair market level after an imbalance-causing swap. As a result, it is entirely plausible that the price at the beginning of a block reflects a manipulated or stale state rather than true market conditions. This leads to the enforcement of swaps against an inaccurate "anchor price".

Consider whether this can be solved at the design level. If it is an assumption that must be made when using such contracts, consider reflecting it in the contract's docstrings.

M-04 Incorrect `slot0` Persistence on `!zeroForOne` Swaps Causes Inconsistent Pricing Logic

In the `AntiSandwichHook` contract, the logic within `_getTargetOutput` introduces an unintended side effect when handling `!zeroForOne` swaps (i.e., swaps from token1 to token0). Specifically, this line is problematic:

```
if (!params.zeroForOne) {
    _lastCheckpoint.state.slot0 = _lastCheckpoint.slot0;
}
```

The intention is to lock the pool price at the beginning-of-block value, ensuring consistent execution pricing. However, this operation modifies `_lastCheckpoint.state.slot0` persistently. Since this storage is not reset later, it means that the next `zeroForOne` swap will base its logic on a `slot0` value from a previous `!zeroForOne` swap instead of using the current, accurate pool state.

This breaks the asymmetry and expected behavior that:

- `zeroForOne = true` swaps (token0 → token1) should behave according to the current `xy=k` curve
- `zeroForOne = false` swaps (token1 → token0) should use the price at the beginning of the block

Example Scenario

1. Three sequential `zeroForOne` swaps occur using current pool prices.
2. A `!zeroForOne` swap happens, modifying `_lastCheckpoint.state.slot0`.
3. The next `zeroForOne` swap uses the now stale `slot0` value from step 2, leading to a miscalculated `targetOutput`.

If the system design intends to handle swaps asymmetrically, consider adjusting the `_getTargetOutput` function so that, whenever the swap direction is `zeroForOne`, it explicitly sets `slot0` to the current state of the pool to ensure that the most recent price is always used.

Low Severity

L-01 Incomplete Docstrings

Throughout the codebase, multiple instances of incomplete docstrings were identified:

- In `IHookEvents.sol`, the `HookSwap`, `HookFee`, `HookModifyLiquidity`, and `HookBonus` events do not have any explanation regarding the purpose of each parameter.
- The same is true for the `Place`, `Fill`, `Cancel`, and `Withdraw` events of the `LimitOrderHook.sol` contract. Moreover, many function parameters and return values are also not documented.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Notes & Additional Information

N-01 Missing Named Parameters in Mappings

Since Solidity 0.8.18, mappings can include named parameters to provide more clarity about their purpose. Named parameters allow mappings to be declared in the form `mapping(KeyType KeyName? => ValueType ValueName?)`. This feature enhances code readability and maintainability.

Within `LimitOrderHook.sol`, multiple instances of mappings without named parameters were identified:

- The `liquidity` state variable
- The `tickLowerLasts` state variable
- The `orders` state variable
- The `orderInfos` state variable

Consider adding named parameters to mappings in order to improve the readability and maintainability of the codebase.

N-02 Custom Errors in `require` Statements

Since Solidity version `0.8.26`, custom error support has been added to `require` statements. While initially this feature was only available through the IR pipeline, Solidity `0.8.27` extended support to the legacy pipeline as well.

Throughout the codebase, multiple instances of `if-revert` statements that could be replaced with `require` statements were identified:

- `LimitOrderHook`
- `LiquidityPenaltyHook`
- `BaseOverrideFee`
- `BaseHook`
- `BaseDynamicFee`
- `BaseDynamicAfterFee`
- `BaseCustomAccounting`

For conciseness and gas savings, consider replacing `if-revert` statements with `require` ones.

N-03 Unused Error

In `LimitOrderHook.sol`, the `AlreadyInitialized` error is unused.

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused errors.

Conclusion

The audited codebase introduces three new hook contracts: `AntiSandwichHook` to protect from sandwich attacks, `LimitOrderHook` for placing limit orders, and `LiquidityPenaltyHook` to handle potential penalties for JIT attacks. In addition, some small changes introduced since the previous release were also included in the scope.

Multiple high- and critical-severity vulnerabilities were identified, mainly related to the nuances of the inner mechanisms of Uniswap v4. Given the fact that fixing these issues requires a substantial refactoring of the current codebase, the Uniswap team is encouraged to fix the outstanding vulnerabilities and then go for another round of audits for the post-fix version of the codebase.

The Uniswap team is appreciated for being responsive and helpful throughout the audit period. The supplied documentation was also enough to provide the audit team with the necessary context.