OpenZeppelin | security

OpenZeppelin Uniswap Hooks v1.1.0 RC 2 Audit

UNISWAP FOUNDATION

July 9, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-06-16 To 2025-06-26
Languages	Solidity

Total Issues	8 (8 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	3 (3 resolved)
Notes & Additional Information	3 (3 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.2 branch of the OpenZeppelin/uniswap-hooks repository at commit 3e9fa22. While the same scope had already been audited on the release-v1.1-rc.1 branch at commit 0879747, due to the number of important issues uncovered and the consequent refactor needed, a simple fix review was deemed insufficient and instead a re-audit was suggested.

Findings that were present in the previous report but that are not present in this report have been resolved during the codebase refactor.

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.

***Update:** All of the fixes for the findings addressed in this report have been merged at commit 67ddcdf in the `main` branch.*

System Overview

`AntiSandwichHook`

The `AntiSandwichHook` contract implements a sandwich-resistant Automated Market Maker (AMM) design intended to mitigate sandwich attacks, where malicious actors exploit transaction ordering within a block to extract value at the expense of honest users. This is achieved by enforcing the condition that no swap is executed at a price more favorable than the one available at the beginning 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 reference. Subsequent swaps within the same block are then compared against this initial checkpoint. For trades in the `!zeroForOne` direction, the hook restricts execution to ensure that no better-than-baseline prices are obtained. When a trade would yield a more favorable output than the checkpoint price allows, the excess tokens are withheld and processed to prevent value extraction by the swapper.

`LiquidityPenaltyHook`

The `LiquidityPenaltyHook` contract is designed to defend Uniswap V4 pools against Just-In-Time (JIT) liquidity attacks. These attacks involve adversaries briefly injecting liquidity immediately before a large trade, collecting fees, and withdrawing liquidity within the same or following block, effectively extracting value without taking on market risk. This behavior harms long-term liquidity providers (LPs) and undermines fair fee distribution.

To combat this, the hook enforces a time-based penalty mechanism based on the block number at which liquidity is added and subsequently removed. When liquidity is removed too soon (before a configurable `blockNumberOffset` has elapsed), a penalty is applied. This penalty takes the form of a fee donation: part (or all) of the collected fees are redirected back to the pool and distributed among the in-range LPs, discouraging abusive short-term liquidity provision.

`LimitOrderHook`

The `LimitOrderHook` contract allows users to express limit orders by creating out-of-range liquidity positions in Uniswap V4 pools. 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 a one-sided limit order at a specific price level (the tick).

Once the pool price crosses that tick (i.e., it becomes in-range), the liquidity is consumed by a swap and the order is considered filled. The hook listens to swaps and, upon detecting a price crossing, it automatically removes the liquidity and mints the received tokens to itself for later withdrawal.

The contract includes the following features:

- **Order Aggregation**: Orders placed at the same tick and in the same direction are grouped into a single order ID. Each participant's liquidity is tracked individually but contributes to a shared pool of proceeds.
- **Order Cancellation**: Users can cancel their unfilled orders via the `cancelOrder` function. If the user is the last remaining LP for that order, any fees earned are returned to them. Otherwise, accrued fees are allocated to the shared order pool, benefitting the remaining participants.
- **Withdrawals**: After an order is filled, participants can claim their proportional share of the output tokens via the `withdraw` function.
- **Fee Sniping Prevention**: To prevent new participants from unfairly claiming previously accrued fees, the contract implements per-user fee checkpoints at the moment liquidity is added. Only fees accrued after a user's checkpoint are considered in their withdrawal.

Differences from v1.0

Apart from adding 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.

Differences from v1.1.0-rc.1

The audited hooks already exist in the release-v1.1.0-rc.1 version of the codebase. However, their implementations have since been refactored to improve the robustness and security of the hooks. Below are the main changes introduced between release-v1.1-rc.1 and the audited release-v1.1-rc.2.

`LiquidityPenaltyHook`

- **Fee Withholding During Additions**: Unlike the previous version, which only applied penalties at the time of liquidity removal, the updated contract withholds accrued fees at the time of liquidity addition if the position had been created recently. These fees are held within the hook itself, disabling premature collection and preventing attackers from circumventing the penalty by repeatedly adding/removing small amounts.
- **Unified Fee Accounting and Settlement**: The new logic unifies fee management by summing both `feeDelta` (generated during removal) and `withheldFees` (from additions) to compute the total amount subject to penalty. This ensures that the entire lifecycle of the position is taken into account and prevents manipulation through fragmented liquidity provision.
- **Fail-Safe for Empty Pools**: If a penalty is due but the pool has zero active liquidity, the hook reverts to avoid donating fees into an empty pool (which could otherwise cause unexpected behavior or result in permanent loss of withheld funds). This enforces economic correctness even in edge cases.

`AntiSandwichHook`

- **Custom Fee Handling via `_handleCollectedFees`**: A major change in the new version is the introduction of a `virtual` function, `_handleCollectedFees`, which delegates the responsibility of processing the excess collected amount to the inheriting contract. This change provides developers with flexibility in determining how excess fees should be treated (whether they should be donated, redistributed, sent to a treasury, or otherwise handled).

- **Input Adjustment for Fixed Output Swaps**: In scenarios where a user specifies an **exact output** (i.e., fixed output swaps), the hook now adjusts the input amount upwards if the execution would have resulted in a better-than-checkpoint price, ensuring that the user pays at least the target price.
- **Explicit Scope of Protection**: It is now clearly documented and enforced that only swaps with `zeroForOne == false` (typically selling token1 for token0) are protected. In the other direction, swaps behave normally under the AMM curve and are not constrained by the checkpoint.

`LimitOrderHook`

- **Checkpoints**: In order to prevent edge-case scenarios where fees are subjected to be stolen and to make a more robust accounting, checkpoints have been added to better track order placement. This change affected how fees are managed when placing orders and when withdrawing.
- **Cancel Cleanup Improvement**: Changes have been introduced in the cancel process to better reflect when the entire liquidity in an order is being removed.

High Severity

H-01 Infinite Loop in Tick Iteration Due to Misaligned Current Tick

The `AntiSandwichHook` contract implements an anti-MEV mechanism by storing a snapshot of the pool state at the beginning of each block. As part of this process, the `_beforeSwap` function iterates over tick indices from the last checkpoint to the current tick to update liquidity and fee data. This iteration is performed in a `for` loop using a fixed `step` equal to the pool's `tickSpacing`, and continues as long as `tick != currentTick`.

However, `currentTick` may not always be aligned with the pool's configured `tickSpacing`. This misalignment occurs naturally due to the dynamics of price movement in the pool, which can cause the current tick to land on any arbitrary value rather than on a multiple of the tick spacing. When this happens, the loop condition `tick != currentTick` will never be satisfied, because the increment or decrement using `tickSpacing` will skip over the misaligned current tick. As a result, the loop becomes infinite, consuming all available gas and rendering the transaction invalid. This creates a denial-of-service (DoS) vector, as users can no longer execute swaps in the affected pool.

Consider modifying the tick iteration logic to compute the step dynamically based on the direction and difference between the current tick and the checkpoint tick, ensuring that the loop reliably reaches the current tick even when it is not aligned with the tick spacing.

Update: *Resolved in pull request #80 at commit 5e42129. The team stated:*

> *In order to solve the infinite loop iteration, we now cache the `lastTick` value before `_lastCheckpoint.state.slot0` is updated and instead of comparing `currentTick != lastTick`, which could cause the misalignment, we check if `currentTick <= lastTick` or `currentTick >= lastTick`. We also added a comment on the natspec with a warning on the possibility of a large tick difference which could lead to a large for loop (although not infinite) which could lead to `MemoryOOG` error in extreme cases (small tick spacing and really large tick difference).*

Medium Severity

M-01 Incorrect Fee Application When `unspecifiedAmount` Represents Input Instead of Output

The `BaseDynamicAfterFee` contract enables dynamic fee enforcement by comparing the swap's `unspecifiedAmount` with a target value and charging the difference as a fee. This logic assumes that `unspecifiedAmount` always represents the output of the swap. However, if the user performs an exact output swap, `unspecifiedAmount` will actually represent the input amount the user must pay (`unspecifiedAmount < 0`).

In cases where `unspecifiedAmount` corresponds to the input, the current implementation incorrectly applies a fee if the input exceeds the target output, leading to users being overcharged. This occurs because the fee is always calculated as `feeAmount = uint128(unspecifiedAmount) - targetOutput`, even when `unspecifiedAmount` is an input. As a result, users may pay unnecessary additional fees unrelated to any received output.

No clear way to exploit this issue has been identified within the current `AntiSandwichHook` implementation. However, since `BaseDynamicAfterFee` is an abstract contract designed to be extended by future hooks, consider modifying the logic in `BaseDynamicAfterFee` to only apply the fee when `unspecifiedAmount` represents the output side of the swap. This ensures that users are not charged fees based on their input amounts and preserves the intended semantics of post-swap fee enforcement.

Update: *Resolved in pull request #86 at commit 2678eb9. The team stated:*

> *We updated the `BaseDynamicAfterFee` logic to differentiate between `exactInput` or `exactOutput` swaps, being more explicit about handling `unspecifiedAmount` instead of only outputs. In order to be more explicit, we renamed `_getTargetOutput` to `_getTargetUnspecified`. In the `AntiSandwichHook` level, we removed the `_handleCollectedFees` functions, leaving the handling of tokens to be written directly on `_afterSwapHandler`.*

Low Severity

L-01 Missing Docstrings

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

- In `BaseHook.sol`, the `poolManager` state variable
- In `LiquidityPenaltyHook.sol`, all state variables

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #85 at commit 933feb7.*

L-02 Liquidity Penalty Can Be Circumvented Using Secondary Accounts

`LiquidityPenaltyHook` is designed to mitigate JIT liquidity attacks by penalizing fee collection when liquidity is added and removed within a short window (`blockNumberOffset`). Fees accrued during this window are redirected to active in-range LPs, thereby not giving incentives to opportunistic liquidity provision around swaps. However, a coordinated attack involving multiple accounts can still be used to bypass this penalty mechanism under specific conditions.

The core of this exploit lies in the ability to manipulate who receives the penalty donation. Because the donated fees are distributed to whoever is in-range at the time of liquidity removal, an attacker can use a secondary account to strategically position liquidity in an otherwise empty tick range. The attack proceeds as follows:

1. **Setup:** The attacker (Account B) provides a small amount of liquidity in a distant or low-traffic tick range where no other liquidity exists.
2. **JIT Execution:** After some blocks, the main attacker account (Account A) adds a large liquidity position around the current tick, anticipating incoming user swaps. These swaps generate fees that accrue to A, but are withheld by the hook due to the JIT window.

3. **Fee Redirection:** After fees are generated, A moves the pool's price into B's tick range via a swap. When A removes liquidity, the hook penalizes the action by donating fees to the in-range position (B's position). B can then immediately withdraw the donated fees.
4. **Optional Reversion:** A can optionally swap again to return the pool to its original state.

Although this strategy is technically feasible, it is rarely practical in real-world conditions. The attack relies on the attacker being able to move the price into a specific tick range, something that becomes significantly more costly and difficult as pool liquidity increases. In highly liquid pools, the cost of such manipulation often outweighs the potential gain from the collected fees. Furthermore, to extract meaningful profit, the attacker would need to intercept a large volume of user swaps within the JIT window, which introduces additional uncertainty and complexity.

Due to these constraints, while the mechanism remains exploitable in theory, it is of low practical viability. The attacker must incur high costs to control price movement and depend on timing a substantial amount of user activity, both of which reduce the profitability and feasibility of the exploit. As such, this issue has been categorized as having a low severity. It highlights a subtle limitation in the fee donation logic of the hook but does not pose a realistic threat under normal market conditions. Nonetheless, developers and protocol designers should remain aware of the potential for fee redirection through coordinated account behavior, especially in low-liquidity pools.

Consider expanding the hook's docstrings to explicitly mention the possibility of coordinated multi-account strategies redirecting penalties, particularly in low-liquidity environments, to ensure that downstream integrators are aware of this edge case.

Update: *Resolved in pull request #89 at commit bd7b885.*

L-03 Misleading Naming in `getTargetOutput` Can Cause Developer Confusion

The `getTargetOutput` function calculates the unspecified amount in a swap based on the beginning-of-block pool state. This value can represent either the input or output of the trade, depending on the swap direction and whether it is an exact input or exact output swap. Despite this, the function name implies it that always returns an output amount, which does not accurately reflect its behavior.

Consider renaming the function to `getTargetUnspecifiedAmount` to more clearly convey that the returned value may be either the input or the output. This would reduce potential confusion for developers and improve the overall clarity and maintainability of the code.

Update: *Resolved in pull request #86. The team stated:*

> *We renamed `_getTargetOutput` to `_getTargetUnspecified`.*

Notes & Additional Information

N-01 Documentation Mismatch in `getLastAddedLiquidityBlock` Function

The comment above the `getLastAddedLiquidityBlock` function within the `LiquidityPenaltyHook` contract incorrectly indicates that it tracks the `withheldFees` for a liquidity position. In reality, the function returns the block number corresponding to the last liquidity addition from `_lastAddedLiquidityBlock`.

This inconsistency between the documentation and the actual code behavior can lead to developer confusion and should be corrected to accurately reflect the function's purpose.

Update: *Resolved in pull request #85 at commit 86facc4.*

N-02 Variables Initialized With Their Default Values

Throughout the codebase, multiple instances of variables being initialized with their default values were identified:

- In `BaseCustomCurve.sol`, the `amount0` and `amount1` variables
- In `LimitOrderHook.sol`, the `amount0` and `amount1` variables

To avoid wasting gas, consider not initializing variables with their default values.

Update: *Resolved in pull request #85 at commit 9cb169c.*

N-03 Missing Return Statement in Withdraw Callback Branch

Within the `unlockCallback` function of the `LimitOrderHook` contract, the branch handling the `Withdraw` callback type decodes the withdrawal data and calls `_handleWithdrawCallback(withdrawData)` but does not return any value. The function is declared to return `bytes memory`, so all branches are expected to return an encoded byte array.

This omission can lead to undefined behavior at runtime and is inconsistent with the function's signature. Consider returning the necessary value in the branch handling the `Withdraw` callback type.

Update: *Resolved in pull request #85 at commit 66231a1. The team stated:*

> *The specific branch doesn't have any value to return, but in order to prevent undefined behavior, we updated the function to return `ZERO_BYTES` in that branch*

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.

After an initial audit of release `v1.1.0-rc.1`, multiple high- and critical-severity vulnerabilities were identified, mainly related to the nuances of the inner mechanisms of Uniswap v4. The codebase was refactored and previously identified issues were tackled, as such the codebase went through another audit, the output of which is this report. The audit identified one high-severity issue, while the previously reported issues have been resolved.

The Uniswap Foundation 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.