OpenZeppelin | security

OpenZeppelin Uniswap Hooks v1.0.0 RC 1 Audit

UNISWAP FOUNDATION

July 3, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-01-07 To 2025-01-17
Languages	Solidity

Total Issues	22 (16 resolved, 2 partially resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	6 (5 resolved, 1 partially resolved)
Low Severity Issues	4 (3 resolved, 1 partially resolved)
Notes & Additional Information	9 (5 resolved)

Scope

We audited the OpenZeppelin/uniswap-hooks repository at commit 1db9646.

In scope were the following files:

```
src/
├── base/
│   ├── BaseCustomAccounting.sol
│   ├── BaseCustomCurve.sol
│   ├── BaseHook.sol
│   └── BaseAsyncSwap.sol (renamed from BaseNoOp.sol)
├── fee/
│   ├── BaseDynamicFee.sol
│   ├── BaseOverrideFee.sol
│   └── DynamicAfterFee.sol
└── lib/
    └── CurrencySettler.sol
```

System Overview

The OpenZeppelin Uniswap Hooks library offers various abstract contracts that can be inherited and used as a base to implement hooks, offering a structured foundation for performing specific actions across one or more Uniswap V4 liquidity pools.

`BaseHook`

This hook consists of the entire hook interface that is only callable from the `PoolManager` contract. In addition, it requires the inheriting contract to specify the permitted hook calls in the `getHookPermissions()` function and override the corresponding internal hook call functions. The last 14 bits of the deployed address need to have the right flags on for each hook permission granted. All of the following hooks inherit this base hook.

`BaseDynamicFee`

This hook consists of a single permission on `afterInitialize` callbacks to update the dynamic LPFee inside the underlying pools to a value specified by the inheriting contract. Inheriting contracts need to override the `_getFee` function in order to return a fee for each supported pool. In addition, anyone can poke pools that have been initialized with this hook contract to update its dynamic LPFee to the value returned by `_getFee`.

`BaseOverrideFee`

This hook has permissions for the `afterInitialize` and `beforeSwap` callbacks. Initialization will be reverted if the pool is not set to have dynamic fees. Before each swap, an overridden fee value is provided to be used as the LPFee for that swap. Inheriting contracts specify the new fee amount in their `_getFee` function.

`DynamicAfterFee`

This hook has the `afterSwap` and `afterSwapReturnDelta` permissions. After each exact-input swap, a fee denominated in the output token is donated to the pool. The amount of the donation is the difference between the output amount and a pre-specified target amount. Inheriting contracts need to have a way to set the target amount for each supported pool before each swap as this hook will reset the target amount each time.

`BaseAsyncSwap`

This hook, renamed from `BaseNoOp`, has the `beforeSwap` and `beforeSwapReturnDelta` permissions. Prior to each exact-input swap, the entire specified amount will be taken by the hook and the swap logic in the underlying Uniswap V4 pool, including charging the LPFee and the protocol fee is completely skipped.

`BaseCustomAccounting`

This hook is expected to be used with only one pool. As a result, it has the permission for the `beforeInitialize` callback, where it initializes its `poolKey` state variable to the pool's address. This hook is meant to own the entire liquidity in the underlying pool and the liquidity modifications are only possible through this hook. In order to achieve that, the hook specifies the permissions for the `beforeAddLiquidity` and `beforeRemoveLiquidity` callbacks, which revert every time when anyone other than the hook itself tries to modify the liquidity through the `PoolManager` contract.

Users can only provide liquidity through the hook directly, by calling either the `addLiquidity` or the `removeLiquidity` functions. Inheriting contracts need to override the `_getAddLiquidity` and `_getRemoveLiquidity` functions which allow for custom calculation logic to convert input amounts to liquidity. Inheriting contracts also need to override the `_mint` and `_burn` functions which allow custom logic to convert Uniswap amounts/ liquidity to hook shares for hook users.

`BaseCustomCurve`

This hook inherits the `BaseCustomAccounting` hook but overrides some of its already implemented internal functions such as `_modifyLiquidity` and `_unlockCallback`. In this hook, when liquidity is being added, the hook calls the `mint` function of the `PoolManager` contract to convert a custom amount of an ERC-20 token (specified by the virtual `_getAmountIn` function) supplied by the caller to underlying ERC-6909 claim tokens in the hook's account and then mints shares for the caller. Similarly, when liquidity is being removed, it calls the `PoolManager.burn` function to burn a custom amount (specified by the virtual `_getAmountOut` function) of ERC-6909 tokens it owns, after which it withdraws the underlying ERC-20 tokens to the caller from the pool and finally burns the shares from the caller.

The inheriting contracts need to put custom calculations and access control for share conversion in the `_mint` and `_burn` functions. In addition, this hook has the `beforeSwap` and `beforeSwapReturnDelta` permissions, which allow the hook to zero out the specified amount for every swap and return a custom unspecified amount via the virtual `_getAmount` function, hence achieving the effect of a custom curve on the swaps. The swaps are settled using the claim tokens owned by the hook contract.

Security Model and Trust Assumptions

The OpenZeppelin Uniswap Hooks library consists mostly of abstract contracts that are not meant to be deployed on their own. As such, the burden of security eventually falls on the inheriting contracts to add access control to sensitive functions, impose guards on external permissionless functions, and implement correct logic in overridden functions.

Integration Guide

There are several general security considerations that inheriting contracts should pay attention to.

Single Pool vs. Multiple Pools

A hook address can be used by any pool without the hook's consent unless it is rejected deliberately. If the hook intends to only interact with a single pool, consider explicitly restricting other pools from using the hook. If the hook intends to support multiple pools, particular care needs to be taken to avoid overriding pool states from multiple allowed pools. This would require the inheriting contract to separate the `poolId` space for the relevant hook states.

Skipped Callbacks

The hooks library skips some permissioned hook callbacks when the caller is the hook itself. For example, if a hook initiates a call to the `PoolManager.modifyLiquidity` function, the `beforeAddLiquidity` and `beforeRemoveLiquidity` hook callbacks will be skipped. If a hook initiates a swap, the `beforeSwap` and `beforeSwapReturnsDelta` callbacks will be skipped. Thus, if the inheriting contract intends to allow initiating calls to the `PoolManager`, some of its permissioned callbacks may be skipped. However, its permissioned callbacks would still be active for other users calling `PoolManager` directly.

Crafting `unlock` Calldata

Hooks that intend to interact with the `poolManager` contain an `unlockCallback` function with user-defined input. This function will be called by the `poolManager` on the hook contract. Depending on the calldata of the unlock function, this could be used to call any function on the hook. Hence, any inheriting contract should limit their users from being able to craft the `unlock` calldata, which could expose unintended functions on the hook.

Hooks Modify Liquidity

When a hook attempts to modify liquidity inside a position in a pool, the returned delta contains fees accrued from that position. This affects the usual signs one would assume when adding liquidity. Hence, any logic processing the returned delta should consider all possible returned values. Furthermore, since the pool's current tick could be updated unfavorably before the liquidity actions, consider also having slippage protection in both cases. If the hook supports native token pools, it should be able to accept and settle native tokens.

Hooks Manage Liquidity

When a hook manages tokens with shares, it is particularly important to differentiate Uniswap liquidity from hook shares. Consider reserving the "liquidity" naming to refer to Uniswap liquidity exclusively and use other terms, such as "shares" to refer to any hook-owned share tokens. If a hook contract owns liquidity, there should exist functions to allow for the removal of locked tokens with either access control or correct accounting.

Dynamic Fee Hooks

Hooks supporting dynamic fees have to implement a function, which allows for setting a new fee for the swaps. If such a function relies on some external factors, which could be temporarily manipulated (e.g., liquidity or the amount of tokens inside a pool), it should have access control so that it is not possible to maliciously manipulate the fee.

Custom Swapping Logic

Since Uniswap V4 supports pools with native tokens, it introduces a possibility of reentrancy, either to the `PoolManager` contract or to the hook itself. These scenarios should be taken into account while designing any custom swapping logic that may depend on the underlying token balance to avoid potential price manipulation.

Specific In-Scope Hook Notes

The `DynamicAfterFee` hook donates the entire after-swap fee to the underlying pool. Since the donations are provided to all the positions including the current tick, it allows arbitragers to provide just-in-time liquidity before the swap to the pool, narrowly around the final price tick after the swap. As such, they would benefit from the donations, although the liquidity they provided was barely involved (or not involved at all) in the swap. This should be taken into account while inheriting from the `DynamicAfterFee` hook.

Contracts inheriting from the `BaseCustomAccounting` hook need to correctly implement the functions responsible for calculating the shares and liquidity modification parameters, such as `_getRemoveLiquidity` and `_getAddLiquidity`. These functions should take into account that the amounts of tokens in the underlying pool may change over time, which can introduce rounding errors.

The `BaseAsyncSwap` and `DynamicAfterFee` hooks only apply the special logic on the exact input swaps. On exact output swaps, they perform normal swaps, according to the Uniswap V4 logic. This could change the price and token amounts inside the underlying pool. If this is not the desired behavior, the inheriting contracts should also apply the custom logic to the exact output swaps.

Critical Severity

C-01 Non-Explicit Multiple-Pool Support Allows Overwriting Hook State

The `BaseCustomAccounting` hook is meant to only support a single UniswapV4 pool defined by the `poolKey` hook state variable. The `poolKey` is initialized in the `_beforeInitialize` hook function when the pool is created and is the only place where the `poolKey` variable may be set.

However, the `poolKey` variable can be overwritten when this hook contract instance is registered for another pool. Due to the underlying asymmetry between pool-hook awareness, anyone can use any valid hook address to any pool without the hook's consent unless it is deliberately rejected by the hook. As a result, all the liquidity provided by users via the hook to the initial pool will be locked, as the `BaseCustomAccounting` contract will now only operate on a newly registered pool, which can be again overwritten in the future.

In the `_beforeInitialize` function, consider ensuring that only one pool can be registered, and revert for any subsequent calls if a hook intends to support only one pool. In addition, for each hook, consider clearly documenting the underlying non-explicit multiple pool support whereby a hook is allowed to be registered for multiple pools. This could help the inheriting contracts be cautious about any hook states being unintentionally overwritten from multiple linked pools.

***Update:** Resolved in pull request #31 at commit 388fcfb.*

High Severity

H-01 Hooks Do Not Support Native Tokens

Uniswap V4 pools support native tokens, such as ETH, without needing to wrap them. The `CurrencySettler` library also supports native-token transfer inside its `settle` function which is called by the hooks whenever there is a need to transfer tokens to the Uniswap V4's

`PoolManager` contract. In the case of native token transfers, the specified amount of native token is transferred using the `payable` `PoolManager.settle` function.

However, none of the hook contracts is capable of receiving native currency. This means that the hook contract will always have 0 native token balance. As a result, pools containing ETH as one of the tokens will not be supported. For example, it will not be possible to provide liquidity to pools through the `BaseCustomAccounting` hook.

Consider allowing hooks to receive correct amounts of native currency so that it can be used for the `settle` calls to the `PoolManager` contract.

***Update:** Resolved in pull request #32 at commit 4be1870.*

H-02 Insufficient Slippage Check

The `BaseCustomAccounting` hook requires users to modify the associated UniswapV4 pool's liquidity only via the hook itself. The liquidity may be modified by using the permissionless external `addLiquidity` and `removeLiquidity` functions which interact with the UniswapV4 `PoolManager`'s `modifyLiquidity` function. The slippage checks on these two functions are insufficient because such liquidity-altering functions can be front-run by swaps which could change the pool's current tick, affecting the amount of `currency0` and `currency1` required or returned for the liquidity action.

In particular, there is no slippage check when removing liquidity. Furthermore, while there is a slippage check when adding liquidity, it also includes the fees accrued from existing positions, as the delta returned by the `PoolManager.modifyLiquidity` function is a sum of the principal delta and the fees accrued. In the case when the fees accrued are greater than the absolute value of the principal delta, the resulting delta amount may be positive. Thus, unsafe casting of the negation would result in a large positive value, entirely bypassing the slippage check.

Consider performing a slippage check when adding or removing liquidity using only the principal delta.

***Update:** Resolved in pull request #33 at commit 6887feb.*

Medium Severity

M-01 Misleading Comments

Throughout the codebase, multiple instances of misleading comments were identified:

- This comment along with this comment suggests that the fee returned by the `_getFee` function should be denominated in hundredths of a percent. However, inside Uniswap V4, the fee is represented in hundredths of a bip, which is a unit 100 times smaller than the one specified in the comment.
- This comment implies that the `onlyValidPools` modifier ensures that a function using it is only callable by valid pools, whereas, in reality, it does not validate `msg.sender`. Instead, it verifies that such a function is called *for* a valid pool.
- This comment states that the `LockFailure` error is thrown when the hook is not unlocked. However, it is not clear what unlocking means in terms of a hook, and the error is thrown when a call inside the `unlockCallback` fails with no error data. The description of the `LockFailure` error could be changed to better reflect this use case.
- This comment suggests that in the `_mint` function, the liquidity units are minted to the sender. However, the `params` argument passed to that function contains the `to` member, which is meant to be used as the liquidity units' recipient instead of the `msg.sender`. The comment could be changed to reflect that the liquidity units are minted to the `params.to` address, or alternatively, that `AddLiquidityParams.to` could be removed if it is not meant to be used.
- This comment and this comment do not mention all the permissions of the underlying hooks.
- The overriding `_getAddLiquidity` function does not return an encoded version of `ModifyLiquidityParams` as specified by the comments above the `_getAddLiquidity` `virtual` function. The same is true for the `_getRemoveLiquidity` function.
- This comment suggests that liquidity is either added or removed from the pool in the `BaseCustomCurve._unlockCallback` function. However, no Uniswap V4 pool liquidity is involved in the process, only the tokens are being sent to and from the `PoolManager` contract.
- This comment states that the delta returned by the `BaseCustomCurve._modifyLiquidity` function originates from the `PoolManager`. However, the delta is determined without any interaction with the `PoolManager`.

- This comment implies that upon exact-input swaps, `amount0` is specified and `amount1` is unspecified. However, this is not necessarily true as it also depends on the value of `zeroToOne`. The same thing is also true for this comment.

Consider correcting the aforementioned comments to improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #34 at commit a40dd63.*

M-02 Dynamic After Swap Fees May Not Work as Intended

The `DynamicAfterFee` contract introduces an after-swap fee, which is calculated based on the swap delta and on the target delta specified before the swap. However, users can easily avoid paying the fee by specifying the desired output token amount, as the fee-charging logic only works for the exact-input swaps.

Furthermore, the target delta is reset after each swap and there is no built-in way of updating it on this contract. There could be some unintended consequences if it is not updated promptly. For example, it is possible to front-run a swap with a tiny input amount, which would result in the entire output amount of the swap made by the subsequent user being consumed as a fee. When the `targetDelta` is updated in a transparent manner before each swap (e.g., in a `beforeSwap` hook), it is possible to set a swap amount to be less than the target delta in order to entirely avoid paying the after-swap fee.

Consider imposing a fee on exact-output swaps as well, possibly via a `beforeSwap` hook function. Moreover, consider providing comprehensive documentation about the target delta, particularly about when and how it should be updated.

Update: *Resolved in pull request #34 at commit 081d92a.*

M-03 Unsafe Casting Due to Accrued Fees

When adding or removing liquidity from the `BaseCustomAccounting` hook, the hook initiates calls to the `PoolManager.modifyLiquidity` function. The returned delta is a sum of both `principalDelta` and `feesAccrued`. Hence, it is a legitimate scenario that when adding liquidity to an existing position, the returned `delta` does not necessarily have negative values for both `amount0` and `amount1`. Depending on the amount of accrued fees as well as the liquidity to be minted, one can end up with `amount0` and `amount1` either being positive or negative. Hence, when one of these amounts is positive, this casting will result in wrong values. Two scenarios may occur:

- The wrongly cast `uint256(int256(-amount))` to be settled can be very large when `amount` is a small positive value, and it may revert when transferring this large amount fails.
- When the accrued fee is large enough (for a large amount of liquidity) and the wrongly cast value can be transferred, this will result in users paying way more instead of receiving a handsome fee.

Consider adding additional logic to accommodate this legitimate scenario without assuming it. This can be achieved by checking the sign of the returned delta amount directly, and calling `take` when it is positive and calling `settle` when it is negative.

***Update:** Resolved in pull request #33 at commit 6d1e94b.*

M-04 `BaseDynamicFee` Hook Can Be Poked Arbitrarily

The `BaseDynamicFee` hook contract has a `virtual` `external` `poke` function that allows any pool initiated with this hook to (re)set an updated LPFee by anyone at any time. The updated LPFee comes from the `_getFee(poolKey)` virtual function, which could return any fee based on either the current pool state or any other external conditions. However, if the `_getFee` implementation depends on some external factors, it opens up the possibility of manipulating the LPFee and swapping in a single transaction.

For example, suppose that `_getFee` for a V4 pool is dependent on the token balance of a Uniswap V3 pool having the same token pair. Suppose also that the returned fee for the V4 pool is inversely correlated to the corresponding V3 pool token balance. That is, `_getFee` returns a lower value when the V3 pool has a large balance and otherwise returns a higher value. Now, an attacker could take a flash loan, deposit into the V3 pool temporarily, and then call `poke` on the V4 pool. Since the V3 pool balance would have increased, the swap fee will drop and the attacker can pay lesser fees for subsequent swaps. If the subsequent users do not `poke` in advance, they can enjoy cheap fees without having to do any manipulation.

Consider imposing access control on the `poke` function so that it cannot be called arbitrarily. Otherwise, the `poke` function can be made `internal`, allowing the inheriting contracts to decide how to appropriately incorporate it with their logic. Alternatively, consider updating the documentation of the `poke` function so that the risk described above is clear to the inheriting contracts.

Update: *Resolved in [pull request #34](#) at commit [7e81272](#).*

M-05 `BaseAsyncSwap` Operation Is Unclear

The [`BaseAsyncSwap` hook](#) skips any swap actions in the Uniswap pool by [minting](#) the corresponding ERC-6909 claim tokens to itself. However, this only applies to exact input swaps. There is no documentation describing how this amount in the hook can be distributed, used, withdrawn, or refunded back to the user. Consider adding more documentation regarding this or indicating a clear intention for `virtual` functions to be overridden by inheriting contracts.

Furthermore, the `BaseAsyncSwap` hook can [receive](#) ERC-6909 tokens on each exact input swap from multiple pools. However, it does not implement any mechanism to separate the same tokens received from different pools for different users. For example, it could record the amount of tokens, the address of the pool, and the address of the user who initiated the swap in order to be able to perform subsequent logic related to the owned tokens.

In addition, the `BaseAsyncSwap` hook can be skipped by specifying the output amount in a swap. Thus, to implement the corresponding logic for exact-output swap, the `beforeSwapDelta` callback needs to return a negative amount to zero out the output amount. In this case, the hook has to settle the corresponding amount. This may require some additional mechanism to correctly account for the inflow and outflow of the tokens involved.

Consider having a clearer utility for the `BaseAsyncSwap` hook so that the inheriting contracts can clearly identify the use case for incorporating their own logic.

Update: *Partially resolved in [pull request #34](#) at commit [1605117](#). The documentation has been extended, but the address of the user who initiates a swap is still not recorded and the hook still works only on exact input swaps. There are plans to modify this contract in the future in order to resolve these concerns.*

M-06 Unintuitive Return Delta

The [`BaseCustomCurve` hook](#) allows users to add and remove liquidity for a particular pool, but only via the hook itself. It also only allows swapping with the available liquidity in the hook. This hook [inherits](#) the `BaseCustomAccounting` and overrides its implementations of the `_modifyLiquidity` and `_unlockCallback` functions in order to reuse the logic in the `addLiquidity` and `removeLiquidity` functions.

The overridden `_unlockCallback` function always returns two negative amounts in its delta regardless of whether it is called from `addLiquidity` or `removeLiquidity`. This behavior is unintuitive and could lead to mistakes when the delta is consumed by the to-be-implemented `_mint` or `_burn` functions. In fact, the `_mint` function expects the returned delta to have the same sign as `poolManager.modifyLiquidity` (i.e., negative amounts if not considering fees) while the `_burn` function expects the returned delta to have positive amounts.

Consider returning positive amounts when removing liquidity to be consistent with the signs expected by the inheriting contracts. This is so that any inheriting contracts can implement the `_mint` and `_burn` functions in a more intuitive way.

Update: *Resolved in pull request #36 at commit 45edf24.*

Low Severity

L-01 Redundant `virtual` Function Modifier

The `validateHookAddress` function of the `BaseHook` contract ensures that the hook has the expected permissions. This function is based on the `validateHookAddress` function from Uniswap's `v4-periphery` repository and has the `virtual` modifier. This allows the contracts inheriting from the `BaseHook` to override the function. However, the `virtual` function modifier was introduced to the `validateHookAddress` function from the `v4-periphery` repository in order to facilitate testing and is not necessary otherwise.

Consider removing the `virtual` modifier from the `validateHookAddress` function in order to improve the readability of the codebase and avoid situations where the function is incorrectly overridden.

Update: *Resolved in pull request #35 at commit 1c1f186.*

L-02 Insufficient Documentation

Throughout the codebase, multiple instances of insufficient documentation were identified:

- The values returned by the `_getAddLiquidity` and `_getRemoveLiquidity` functions of the `BaseCustomAccounting` contract are passed to the `PoolManager.modifyLiquidity` function as arguments. In order to prevent users from being able to withdraw liquidity specified by different users, these arguments (specifically, the `ModifyLiquidityParams` object) should contain the salt uniquely set for each liquidity provider. Otherwise, when the liquidity is removed, anyone could specify the liquidity to remove and the tick range, and as long as they have enough liquidity units inside the hook, they could remove liquidity provided by someone else, possibly stealing the accrued fees. This fact could be documented in the documentation for the `_getAddLiquidity` and `_getRemoveLiquidity` functions.

- The `DynamicAfterFee` contract allows for imposing an additional fee which will be accounted for after the swaps. The fee is based on the target and actual tokens' delta values and is donated to the underlying pool. Whenever a donation is made to the pool, it is awarded for all the in-range liquidity providers. It could be exploited by an attacker, who could provide just-in-time liquidity to a narrow range around the final tick after the swap. Such liquidity would barely be involved (or not involved at all) in a swap, but the attacker would receive a huge part of the donation. Consider documenting this possibility, perhaps in a way similar to how it is done in Uniswap V4.
- The `BaseCustomCurve` contract would benefit from additional documentation. In particular, the `take` and `settle` operations inside the `_beforeSwap` and `_unlockCallback` functions could be explained in more detail so that the operations they perform are easier to understand.
- The `unlockCallback` function of the `BaseHook` contract does not provide any documentation for its returned value. Furthermore, several `internal` functions, such as the `_getAmountOutFromExactInput` function of the `BaseCustomCurve`, contract do not document their parameters or their return values.
- The intended use cases for the `BaseAsyncSwap` and `DynamicAfterFee` hooks are not included in their documentation, and in order to use them properly, the implementing contracts will need to define additional functions. Consider documenting the intended use cases of these contracts, possibly providing a tangible example of how they could be used. Moreover, consider providing example implementations of the functions that will need to be implemented by inheriting contracts.

Consider fixing all the instances of insufficient documentation listed above in order to improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #35 at commit 315e0e4.*

L-03 Insufficient Arguments

As all the hooks from the library are meant to be general-purpose template hooks, some `virtual` functions from these hooks that are meant to be overridden by inheriting contracts can benefit from having more arguments. Some examples are:

- In the `BaseDynamicFee` hook, the `_getFee` `virtual` function could have `currentTick` and `sqrtPricex96` parameters.
- The `feesAccrued` delta returned from the `PoolManager.modifyLiquidity` call could be returned by the `_modifyLiquidity` function and passed on to the `_mint` and `_burn` `virtual` functions.
- In the `BaseCustomAccounting` hook, a `salt` value could be added to the `AddLiquidityParams` struct so that it could be passed to the `_getAddLiquidity` function in order to allow multiple positions on the same tick range for the same user. Similarly, a `salt` member could be added to the `RemoveLiquidityParams` struct.

Consider implementing the changes suggested above in order to provide a broader and more general functionality for the hooks.

Update: *Partially resolved in pull request #35 at commit 4be1870. The first suggestion hasn't been implemented as the `currentTick` and `sqrtPricex96` values can be fetched from the pool.*

L-04 BaseHook's `unlockCallback` Not Needed

A minimal hook does not need to have an `unlockCallback` function since only the hooks intending to initiate calls to the `PoolManager` contract would need to unlock it. Hence, it is not necessary to have the `unlockCallback` function and the `internal` `_unlockCallback` function in the `BaseHook` implementation.

In addition, the `_unlockCallback` function uses a generic low-level call and is overridden by all contracts in scope, as it is hard to use it without utilities for encoding/decoding arguments to interact with the Pool Manager. Furthermore, leaving the current default `unlockCallback` implementation may have unintended consequences of opening up a way to re-enter the hook for inheriting contracts.

Consider removing the `unlockCallback` function from the `BaseHook` contract and instead allowing the inheriting contracts to add this function if needed.

Update: *Resolved in pull request #35 at commit 9ee7d09.*

Notes & Additional Information

N-01 Typographical Errors

Throughout the codebase, multiple instances of typographical errors were identified:

- In line 22 of `BaseCustomAccounting.sol`, "implementator" should be "implementer" and "Aditionally" should be "Additionally".
- In line 40 of `BaseCustomAccounting.sol`, "An" should be changed to "A".
- In line 13 of `CurrencySettler.sol`, "make" should be changed to "may" or "may make".

Consider correcting all instances of typographical errors in order to improve the clarity and readability of the codebase.

Update: *Resolved in pull request #36 at commit 130875c.*

N-02 Non-Existent Named Return Variable

The documentation of the `unlockCallback` function refers to the `delta` return parameter, but such a return parameter does not exist in this function.

Consider introducing a return parameter named `delta` to the `unlockCallback` function in order to comply with the documentation.

Update: *Resolved in pull request #35 at commit 9ee7d09. This item was addressed by removing `unlockCallback` in finding L-04.*

N-03 Optimizable State Reads

In the `BaseCustomCurve` contract, the `poolKey` storage reads could be optimized by caching the `poolKey` and operating on a `memory` object instead.

Consider reducing unnecessary SLOAD operations and saving gas by caching the value of the `poolKey` variable in a `memory` variable.

Update: *Resolved in [pull request #36](#) at commit [24d8c6c](#).*

N-04 Unused Modifier

The `onlySelf` modifier that is defined in the `BaseHook` contract is not used anywhere in the codebase.

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

Update: *Acknowledged, not resolved. The team stated:*

> This was implemented as helpers, similar to the functionality found in `onlyValidPools`.

N-05 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, none of the contracts has a security contact.

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the [OpenZeppelin Wizard](#) and the [ethereum-lists](#).

Update: *Acknowledged, not resolved. The team stated:*

> Implementers should add their own security contacts.

N-06 Different Pragma Version From `PoolManager`

The all the base contracts in scope use a floating pragma (`^0.8.24`).

Consider having a fixed pragma version that is the same as the one used in the `PoolManager` contract, which is 0.8.26. This will help improve consistency and avoid unexpected compiler behavior.

Update: *Acknowledged, not resolved. The team stated:*

> This codebase is using `IPoolManager` which has ^0.8.24, matching our pragma versions.

N-07 Naming Suggestions

Throughout the codebase, multiple opportunities for improved naming were identified:

- The manager variables inside the `CurrencySettler` library could be renamed to `poolManager`, as that name is used to refer to the Uniswap V4 monolithic `PoolManager` contract in other in-scope contracts.
- The `DynamicAfterFee` contract name could be changed to `BaseDynamicAfterFee` to better reflect the fact that it refers to an abstract contract and to be consistent with other abstract contracts' names.
- The variables named `liquidity` related to the internal hook's liquidity units could be renamed to `shares` and the relevant comments updated in order to make a clear distinction from the Uniswap V4 liquidity.
- The `amountIn` parameter of the `_getAmount` function refers to the amount of the specified token, which does not necessarily have to be the input token. In fact, the same parameter is called `amountOut` in the `_getAmountInForExactOutput` function. Consider renaming the `amountIn` parameter of the `_getAmount` function to `amountSpecified`.

Consider implementing the above-mentioned naming suggestions to improve the readability and clarity of the codebase.

Update: *Resolved in pull request #36 at commit 9cef475.*

N-08 Functions Are Updating the State Without Event Emissions

Throughout the codebase, multiple instances of functions updating the state without an event emission were identified:

- The `_beforeInitialize` function in `BaseCustomAccounting.sol`.
- The `constructor` function in `BaseHook.sol`.

Consider emitting events whenever there are state changes to improve the readability of the codebase and make it less error-prone.

Update: *Acknowledged, not resolved. The team stated:*

> This will not be fixed in this codebase in order to allow implementers to add their own events, if necessary.

N-09 Unnecessary Restriction On the `_getAmount` `virtual` Function

The `_getAmount` function of the `BaseCustomCurve` hook contains the entire logic for calculating the swap amounts, which should also include logic for the fee mechanism, input amount, token computation, swap direction, etc. Thus, this function could be complex in its general form. However, the impact of the parameters and internal restrictions on how to implement this `virtual` function may cause unnecessary confusion.

The parameters of the `_getAmount` function are not succinct. If the `input` and `output` currencies are specified, there is no need to specify `zeroForOne`. In fact, it is not necessary to specify `input` and `output` currencies at all as this contract, by inheriting the `BaseCustomAccounting`, is meant to only work with one Uniswap V4 pool. Thus, it is enough to only have the `zeroForOne` parameter to determine the swap direction. It would be enough to have all the original swap parameters passed in. Furthermore, it is unnecessary to impose restrictions on the logic by splitting the function based on `exactInput`. As a complex function, the inheriting contracts may need to break it down anyway according to how it suits the implementation.

Consider passing in the original `IPoolManager.SwapParams` directly to the `_getAmount` function and remove the unnecessary restrictions (i.e., the `_getAmountOutFromExactInput` and `_getAmountInForExactOutput` `virtual` functions). Consider also documenting the need for fee mechanism since the swapping needs to happen inside the hook.

Update: *Resolved in [pull request #36](#) at commit [31a8430](#).*

Conclusion

The audited codebase provides base contracts that can be inherited to implement customized hooks for UniswapV4 pools. As such, it is important for the inheriting contracts to implement all the required `virtual` functions correctly. The guidelines and potential risk areas have been included in the introduction of this report.

The codebase could benefit from further development and additional documentation aimed at improving the utility, clarity, and security of the current and any additional hook archetypes. In particular, a more comprehensive testing suite could be implemented in order to test a wider range of interactions between the hooks and the PoolManager. Furthermore, the `BaseAsyncSwap` and `DynamicAfterFee` hook contracts could be extended with an additional code, responsible for handling all use cases, such as swaps specifying the exact output amount, in order to provide a complete base functionality for the inheriting contracts. Finally, example practical hook implementations utilizing the base hook contracts could be provided in order to demonstrate their potential use cases and the way on how their virtual functions could be overridden in a safe way.