OpenZeppelin | security

Uniswap v4 Core Audit

Uniswap

July 17, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-05-27 To 2024-06-21
Languages	Solidity, Yul

Total Issues	24 (18 resolved, 2 partially resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	3 (2 resolved)
Low Severity Issues	3 (2 resolved)
Notes & Additional Information	16 (12 resolved, 2 partially resolved)
Client Reported Issues	1 (1 resolved)

Scope

We audited the Uniswap/v4-core repository at commit d5d4957.

The following files were new and therefore were fully audited:

```
src
│   ERC6909.sol
│   ERC6909Claims.sol
│   Extsload.sol
│   Exttload.sol
│   NoDelegateCall.sol
│   PoolManager.sol
│   ProtocolFees.sol
├── interfaces/
│   ├── IHooks.sol
│   ├── IExtsload.sol
│   ├── IExttload.sol
│   ├── IPoolManager.sol
│   ├── IProtocolFeeController.sol
│   ├── IProtocolFees.sol
│   ├── external/IERC20Minimal.sol
│   ├── external/IERC6909Claims.sol
│   └── callback/IUnlockCallback.sol
├── libraries/
│   ├── CurrencyDelta.sol
│   ├── CustomRevert.sol
│   ├── FixedPoint128.sol
│   ├── FixedPoint96.sol
│   ├── Hooks.sol
│   ├── LPFeeLibrary.sol
│   ├── LiquidityMath.sol
│   ├── Lock.sol
│   ├── NonZeroDeltaCount.sol
│   ├── ParseBytes.sol
│   ├── Pool.sol
│   ├── ProtocolFeeLibrary.sol
│   ├── Reserves.sol
│   ├── SafeCast.sol
│   ├── StateLibrary.sol
│   ├── TransientStateLibrary.sol
│   └── UnsafeMath.sol
├── types/
│   ├── BalanceDelta.sol
│   ├── BeforeSwapDelta.sol
│   ├── Currency.sol
│   ├── PoolId.sol
│   ├── PoolKey.sol
│   └── Slot0.sol
```

On the other hand, the following files were checked only for their difference against the version present in the Uniswap/v3-core repository at commit d8b1c63.

```
├── libraries/
│   ├── BitMath.sol
│   ├── FullMath.sol
│   ├── Position.sol
│   ├── SqrtPriceMath.sol
│   ├── SwapMath.sol
│   ├── TickBitmap.sol
│   └── TickMath.sol
```

System Overview

Uniswap v4 is an automated market maker (AMM) which uses the concentrated liquidity model by default but can be customized to use any other model. Such customizations are possible because of the hooks that are executed before and after every user action. Other significant changes compared to v3 include singleton architecture, flash accounting, dynamic liquidity providers fee, donation to liquidity providers, and native token support. Finally, the new version uses several techniques to reduce gas usage. The entry point to the Uniswap v4 core is the singleton `PoolManager` contract which contains all the pools within itself. This contrasts with the previous versions of the protocol which had the factory contract deploying individual pools as separate contracts.

Hooks are functions of the `Hooks` contract which is specified at the pool initialization time. These functions are called by the `PoolManager` before and after each swap, position modification, donation, and pool initialization. The available hooks are `beforeInitialize` and `afterInitialize`, which are called during pool initialization, `beforeAddLiquidity`, `afterAddLiquidity`, `beforeRemoveLiquidity` and `afterRemoveLiquidity` which are called when changing liquidity positions, `beforeDonate` and `afterDonate`, which are called when donating assets to LPs, and `beforeSwap` and `afterSwap`, which are called when performing swaps. Some hooks, namely `beforeSwap`, `afterSwap`, and `afterModifyLiquidity`, return parameters which might influence the `PoolManager`'s control flow. The lower bits of the hook address decide which hooks are enabled. Thus, to have a particular set of hooks enabled, one must brute-force the address whose lower bits are the desired bit sequence.

Flash accounting means that all operations within the protocol are performed on transient storage and should be settled only at the end of the transaction. Settlement can be accomplished by either ERC-20 token transfers or ERC-6909 accounting. The latter allows users to keep their token inside Uniswap v4 for future use which is cheaper in terms of gas compared to ERC-20 transfers. One consequence of the flash accounting mechanism is free flash loans. Furthermore, since all the pools reside in the singleton contract, it is possible to flash borrow the whole balance of a token at once.

The liquidity provider fee is now more customizable and can be of either static or dynamic type which is specified at the pool initialization time. Both fees can take values from 0 to 100% with a precision of 0.01%. The static fee is set once and cannot be changed whereas the dynamic fee can be changed by the hook at any time, either for all transactions or per transaction. Note that a hook might charge its own fees from users and liquidity providers on swaps and position modifications which is done via a separate mechanism. The donate function can be used to send tokens to the singleton contract and distribute those to the liquidity providers whose liquidity is currently in use. Native tokens like ETH are supported, allowing users to avoid wrapping and unwrapping tokens. Gas optimizations include the already mentioned use of transient storage as well as more widespread techniques like extensive use of assembly, custom packed types (both in-storage and in-memory), low-level memory management and unchecked math.

Whether it is adding or removing liquidity, doing swaps, or donating, the user always needs to call the `unlock` function first. This function will set an unlock variable to `true` and will call back the `msg.sender` at a specified function. This callback is what is actually used by the caller to call either `modifyLiquidity`, `donate` or `swap`. Since these operations might produce changes in the amount of assets owed to the protocol or the user through the use of balance deltas, once the callback is done, the `unlock` function will perform a check on those deltas and will make sure that they have been zeroed out before finishing the transaction. In this way, the protocol ensures that no debt or credit is left behind.

Because of all this, the `take` and `settle` functions are needed. The `take` function will withdraw assets from the protocol so that any asset owed to the user is given to them. Conversely, the `settle` function will ensure that the user transfers to the protocol all the assets owed to it. If one does not want to `take` or `settle`, the ERC-6909 `mint` and `burn` functions can be used instead with the same desired effect of balancing the deltas.

Finally, the `sync` function is used to update the current protocol balance of any particular asset. This is necessary to be performed before transferring tokens to the protocol and calling the `settle` function.

Security Model and Trust Assumptions

The following observations were made regarding the security model and trust assumptions of the audited codebase:

- The core contract does not have slippage protection and assumes that the periphery contracts implement it. Hence, having slippage protection on the periphery contracts is essential.
- ERC-20 tokens with non-standard implementation including rebasing tokens, double entry point tokens, and so forth or outright malicious tokens are not supported by the protocol and will lead to vulnerabilities in the pools they are used in.
- Malicious hooks might steal tokens from users and liquidity providers in multiple ways. A variety of user risks can be mitigated by implementing slippage protection on periphery contracts, but liquidity provider risks are not mitigated in the same manner. Thus, liquidity providers should examine hooks more closely and put more trust into them. Even though hooks are set at initialization time, they might be upgradeable, and in this case, be able to change their code at later point in time.
- Since Uniswap v4 is permissionless anyone can create a pool with any token. It is up to the periphery contracts, users and liquidity providers to examine and decide which pools they want to use. The core contract, however, limits the impact of malicious tokens and malicious hooks on the pool they are used in by ensuring that each function operates only within the particular pool and separating the accounting of each pool in storage, separating also the balance deltas produced by hooks and by users.

Privileged Roles

On the protocol level, there are two privileged roles which are trusted to not behave maliciously: the protocol fee controller and the owner. The former can set the protocol fee on any pool but the fee is capped at 0.1%. It also can collect accumulated protocol fees to an arbitrary address. The latter can change the protocol fee controller.

On the individual pool level, the hook can change the liquidity provider fee but only if the pool was initialized with the dynamic fee in the first place.

Critical Severity

C-01 ERC-20 Representation of Native Currency Can Be Used to Drain Native Currency Pools

The `settle` function, responsible for settling a user's debt, increases the account delta of the specified currency. There are two settlement flows: one for the native currency and another for all other currencies. If the currency is native, the amount used for increasing the delta is `msg.value`. Otherwise, if the currency is a regular ERC-20 token, the amount is the balance difference between the last time `sync` or `settle` were called and the current `settle` invocation.

This implementation is vulnerable to attacks if the given token has two addresses. This is particularly dangerous since the protocol supports native tokens and operates on chains where the native token has a corresponding ERC-20 token. Examples of such chains include Celo with CELO, Polygon with MATIC, and zkSync Era with ETH. There is a caveat concerning MATIC on Polygon and ETH on zkSync Era: while it is possible to manipulate balance deltas, we have not found a way to withdraw them from the pool since the former has the ERC-20 transfer function that fails if `msg.value != value` and the latter does not have the ERC-20 transfer function at all. However, there is a way to withdraw CELO on Celo from the pool.

To demonstrate the vulnerability, we will use the CELO token as an example. The attack consists of two phases: manipulate the balance deltas and take the tokens out of the pool using these deltas.

The attacker starts with 1000 CELO. To manipulate the balance deltas to their advantage, they would perform the following steps within a single transaction.

1. Call `manager.sync(0x471ece3750da237f93b8e339c536989b8978a438)`, where `0x471ece3750da237f93b8e339c536989b8978a438` is the address of the CELO token and `manager` is the `PoolManager` contract. This loads the current CELO balance of `manager` to transient storage.
2. Call `manager.settle{value: 1000}(address(0x0))`, which increases the `address(0x0)` balance delta of the attacker by 1000 and transfers 1000 CELO to `manager`.

3. Call `manager.settle(0x471EcE3750Da237f93B8E339c536989b8978a438)`, which increases the `0x471EcE3750Da237f93B8E339c536989b8978a438` balance delta of the attacker by 1000 without any need of transferring tokens since the balance of the CELO token has increased in step 2.

At this point, the attacker has increased their balance deltas by 2000 for 1000 CELO tokens. To take out the profit, they would perform the following steps within the same transaction.

1. Call `manager.take(0x471EcE3750Da237f93B8E339c536989b8978a438, address(0x1337), 1000)`, where `address(0x1337)` is the attacker's address. This decreases the `0x471EcE3750Da237f93B8E339c536989b8978a438` balance delta by 1000 and transfers 1000 CELO tokens to the attacker via the ERC-20 `transfer` function.
2. Call `manager.take(address(0x0), address(0x1337), 1000)`, which decreases the `address(0x0)` balance delta by 1000 and transfers 1000 CELO tokens to the attacker via the value attached to the call. Even though there might be no pool with the native currency, this step still works because the `take` function just performs a call with a value attached. Since the CELO balance is there, the call succeeds.

The attacker has 2000 CELO and zero balance deltas, allowing them to finish the transaction with a profit of 1000 CELO. By repeating the steps above, it is possible to completely drain the native currency pool.

Note that, at first glance, a native token with an ERC-20 representation might look like a double entry point ERC-20 token from the perspective of this issue. However, there is an important difference: double entry point ERC-20 tokens, along with other non-standard ERC-20 tokens, are not supported by Uniswap v4, as explicitly stated by the Uniswap team. Native tokens, however, are supported since they are considered safe and are preferred due to gas savings.

Consider changing the way native currency pools work on chains where the native currency has a corresponding ERC-20 token. For example, make the `NATIVE` variable immutable and set it to the ERC-20 token address for chains where native currency has a corresponding ERC-20 token.

Update: *Resolved in pull request #779. The team re-worked how `sync` and `settle` work. Whenever a `sync` is called, the very next `settle` call will settle the previously synced currency, otherwise the transaction will revert. This makes this attack impossible now, because one wouldn't be able to settle the native currency (point 2) before settling the corresponding ERC-20 version (point 3).*

Medium Severity

M-01 Unsafe Assembly Blocks

Throughout the codebase, some assembly blocks that are marked as safe do not follow the Solidity memory model outlined in the Memory Safety section of the Solidity documentation. This might lead to incorrect and undefined behavior.

- In `Hooks.sol`, on lines 138-139, the return data can be longer than 64 bytes, thus potentially overriding the free memory pointer. This might happen if hooks return more than 64 bytes. For example, the `beforeSwap` hook returns three values, which means that it will require `32 * 3 = 96 bytes` as ABI-encoding allocates 32 bytes for each value, which exceeds the scratch space.
- In `TickBitmap.sol`, on lines 55-60, the error signature and parameters take 96 bytes, which is more than the scratch space.
- In `CustomRevert.sol`, on lines 45-63, the error signature and parameters take 96 bytes, which is more than the scratch space.
- In `Currency.sol`, on lines 52-53, the top 4 bytes of the free memory pointer are overridden.

Consider removing the `memory-safe` annotations from assembly blocks that do not follow the Solidity memory model. Alternatively, consider adjusting the assembly blocks to follow the memory model.

Update: *Resolved in pull request #759.*

M-02 `ProtocolFeeController` Gas Griefing

The `ProtocolFees` contract implements the logic of retrieving a fee from the `protocolFeeController` external contract. The call to `protocolFeeController` is limited by the defined `controllerGasLimit` value. In case the call fails (e.g., by consuming all the gas), the returned value of the fee is 0. However, a malicious `protocolFeeController` can execute a gas griefing attack by returning a large amount of data that will be implicitly loaded into memory, thereby significantly increasing the gas cost or even preventing the transaction from executing. Considering how important it is for the Uniswap team to build unstoppable and permissionless code, this might represent a threat to the protocol.

Consider using assembly to execute the call and manually copy only 32 bytes from the return data to memory.

Update: *Resolved in pull request #771.*

M-03 Front-Running Pool's Initialization or Initial Deposit Can Lead to Draining Initial Liquidity

The pool's `sqrtPriceX96` can be manipulated either by front-running the initialization of the pool or by an initial deposit of liquidity. The initialize function allows initializing a pool whose stored ID is obtained by conversion of the PoolKey struct that is passed as an argument in the function call. Since the `sqrtPriceX96` is never used for the ID calculation, anyone can front-run the initialization function call and provide their own value for `sqrtPriceX96`, which will then be used for setting the tick.

Another way to manipulate the `sqrtPriceX96` is by front-running the initial deposit of a recently initialized pool. The `swap` function does not check for liquidity before a swap occurs and, if this happens, the `sqrtPriceX96` value stored will change. Consequently, a malicious actor can move the `sqrtPriceX96` of the pool to any value if the current position has zero liquidity. As a result, to take advantage of the LP's liquidity, an attacker could manipulate the `sqrtPriceX96` right after the pool initialization and before any deposit is done, establishing a different price in the pool that does not reflect the market price, thereby opening it up to arbitrage opportunities.

The aforementioned two cases lead to an exploitation scenario that allows draining the initial liquidity if it is performed on a separate transaction from the one that initializes the pool. In the case of front-running the initial liquidity deposit, consider the following scenario:

1. The LP creates a new pool and sets `sqrtPriceX96` to `79228162514264337593543950336` (`1 tokenA` per `1 tokenB`).
2. The LP calls `modifyLiquidity` to add liquidity to the pool for the price range `0.5 <=> 3`.
3. The attacker calls the swap function to move the `sqrtPriceX96` to a random high value before the LP's `modifyLiquidity` transaction is executed.
4. The LP's `modifyLiquidity` transaction is then executed and liquidity is added.
5. Now, the attacker can swap `1 tokenA` for `3 tokenB` using the LP's liquidity.

In the case of front-running the pool's initialization, one could front-run step 1 of the above scenario so that the LP creation fails (because it will be already initialized by the front-runner) and then skip step 3. This will lead to the same result.

Consider documenting this issue, specifically, the fact that the initialization of a pool can be front-run. In addition, consider whether it should be allowed for a swap to occur with no liquidity at all in the pool or whether it makes sense to provide initialization with initial liquidity within the same transaction. Under the assumption that the `PoolManager` contract will be used through periphery contracts, this issue is not easy to exploit. However, such an assumption might not hold in some circumstances, making the issue more severe.

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

> *This attack vector is prevented by peripheral contracts adding slippage protection when users add liquidity to a pool.*

Low Severity

L-01 Unsafe ABI Encoding

It is not uncommon to use `abi.encodeWithSignature` or `abi.encodeWithSelector` to generate calldata for a low-level call. However, the first option is not typo-safe and the second option is not type-safe. As such, both of these methods are error-prone and ought to be considered unsafe.

Within `Hooks.sol`, there are multiple uses of unsafe ABI encodings:

- The use of `abi.encodeWithSelector` within `beforeInitialize` function
- The use of `abi.encodeWithSelector` within `afterInitialize` function
- The use of `abi.encodeWithSelector` within `beforeModifyLiquidity` function
- The use of `abi.encodeWithSelector` within `beforeModifyLiquidity` function
- The use of `abi.encodeWithSelector` within `beforeDonate` function
- The use of `abi.encodeWithSelector` within `afterDonate` function

Consider replacing all the occurrences of unsafe ABI encodings with `abi.encodeCall` which checks whether the supplied values actually match the types expected by the called function and also avoids errors caused by typos.

Update: *Resolved in pull request #770.*

L-02 Missing Error Messages in `require` Statements

Throughout the codebase, there are `require` statements that lack error messages:

- The `require` statement in line 14 of `BitMath.sol`
- The `require` statement in line 56 of `BitMath.sol`
- The `require` statement in line 30 of `FullMath.sol`
- The `require` statement in line 113 of `FullMath.sol`

To improve overall code clarity and facilitate troubleshooting, consider including specific, informative error messages in `require` statements. Alternatively, consider using custom errors to maintain consistency throughout the codebase.

Update: *Acknowledged, not resolved.*

L-03 Unsafe Casting

`CurrencyLibrary` creates currency by using the `fromId` function which creates a currency address from a `uint256` value. This might lead to unsafe casting within the `mint` or `burn` functions. While this feature might be interesting as it allows traders to create some form of namespaces for the tokens, it could lead to issues during third-party integrations. The corresponding `toId` function converts the currency into an ID by casting the currency address into a `uint256`. This means that while the functions `fromId` and `toId` are expected to be inverse functions, they are not, since the upper 12 bytes are lost during the conversion to currency in the `fromId` function.

Consider masking the upper bits in the `mint` and `burn` functions to ensure the `uint256` ID value fits into `uint160`. Alternatively, consider thoroughly documenting this behavior.

Update: *Resolved in pull request #776. The team decided to add a bit masking to the `fromId` function so that upper 12 bytes are always shaved off from the id.*

Notes & Additional Information

N-01 Inconsistent Use of Multiple Solidity Versions

The protocol is inconsistently using multiple Solidity versions. As such, the following issues have been identified:

- There are multiple contracts with floating pragmas such as in `ERC6909.sol`.
- There are pragma statements that use an outdated version of Solidity such as in `Extsload.sol` and `IExtsload.sol`.
- There are contracts such as `CustomRevert.sol` or `ERC6909` that use a pragma statement that spans several minor Solidity versions. This can lead to unpredictable behavior due to differences in features, bug fixes, deprecations, and compatibility between minor versions.

Consider pinning the Solidity version more specifically throughout the codebase to ensure predictable behavior and maintain compatibility across various compilers. It is recommended to take advantage of the latest Solidity version to improve the overall readability and security of the codebase. Regardless of which version of Solidity is used, consistently pin the version throughout the codebase to prevent bugs caused by incompatible future releases.

Update: *Resolved in pull request #784. The team addressed standardization. The code has been changed to reflect version `^0.8.0` for any imported contract, library or interface in order to allow integrators flexibility. Contracts that rely on transient storage are marked with version `^0.8.24` while `PoolManager`, which will be the main contract, has been set to a specific `0.8.26` version.*

N-02 Lack of `memory-safe` Annotation in Assembly Blocks

In the codebase, there is extensive use of the `memory-safe` annotation for assembly blocks. However, the `BalanceDelta` and `BeforeSwapDelta` types are missing this annotation.

Consider being consistent with the use of the `memory-safe` annotation.

Update: *Resolved in pull request #778.*

N-03 Discrepancy Between Implementation and Specification of ERC-6909

There is a discrepancy between the implementation of the ERC-6909 `transferFrom` function and its specification. In particular, the specification states that it MUST revert when the caller is not an operator for the `sender` and the caller's allowance for the token `id` for the `sender` is insufficient. This implies that if `sender` is `msg.sender` and they did not set themselves as an operator then `transferFrom` should revert. However, this is not the case as `transferFrom` skips operator and allowance checks in such a case.

Consider adjusting the implementation to follow the specification. Alternatively, consider adjusting the specification if this is a desirable property.

Update: *The team reached out to the EIP-6909 creators and a fix was pushed to the specitifcation.*

N-04 Code Clarity

In the `Pool` contract, there are several places which might benefit from slight refactoring.

- `!exactInput` can be turned into just `exactInput` if the `true` and `false` branches are swapped.
- `!zeroForOne` can be turned into just `zeroForOne` if the `true` and `false` branches are swapped.
- `state.amountCalculated` can use compound operators to shorten the expressions.

Consider refactoring the above to improve code readability.

Update: *Partially resolved in pull request #777. Only the third item has been addressed.*

N-05 Missing Function Parameters Names

Multiple functions declared in the `IProtocolFees` interface are missing parameter names. This makes the code less clear and more difficult to understand.

Consider naming all function parameters.

Update: *Resolved in pull request #772.*

N-06 Discrepancies Between Interfaces And Implementation Contracts

Throughout the codebase, multiple discrepancies have been identified between interfaces and implementation contracts.

- Within the `IExtsload` interface, the first parameter of the `extsload` function is called `slot` whereas in the implementation contract, `startSlot` is used.
- Within the `IERC6909Claims` interface, the first parameter of the `setOperator` function is called `spender` whereas in the implementation contract, `operator` is used.
- Within the `IPoolManager` interface, the parameter of the `settle` function is called `token` whereas in the implementation contract, `currency` is used.

Consider using the same parameter names across interfaces and implementation contracts.

Update: *Resolved in pull request #762.*

N-07 Unused Named Return Variables

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as that function's output. They are an alternative to explicit in-line `return` statements.

Throughout the codebase, there are unused named return variables:

- The `delta` return variable in the `callHookWithReturnDelta` function in `Hooks.sol`
- The `sqrtQX96` return variable in the `getNextSqrtPriceFromInput` function in `SqrtPriceMath.sol`
- The `sqrtQX96` return variable in the `getNextSqrtPriceFromOutput` function in `SqrtPriceMath.sol`
- The `amount0` return variable in the `getAmount0Delta` function in `SqrtPriceMath.sol`
- The `amount0` return variable in the secondary `getAmount0Delta` function in `SqrtPriceMath.sol`

- The `amount1` return variable in the `getAmount1Delta` function in `SqrtPriceMath.sol`
- The `slot` return variable in the `_getPositionInfoSlot` function in `StateLibrary.sol`

Consider either using or removing any unused named return variables.

***Update:** Resolved in pull request #758.*

N-08 Unused Imports

Throughout the codebase, there are imports that are unused and could be removed:

- The import `import {BalanceDelta} from "./BalanceDelta.sol";` imports unused alias `BalanceDelta` in `BeforeSwapDelta.sol`.
- The import `import {Pool} from "../libraries/Pool.sol";` imports unused alias `Pool` in `IPoolManager.sol`.
- The import `import {Position} from "../libraries/Position.sol";` imports unused alias `Position` in `IPoolManager.sol`.
- The import `import {PoolKey} from "../types/PoolKey.sol";` imports unused alias `PoolKey` in `LPFeeLibrary.sol`.
- The import `import {IHooks} from "../interfaces/IHooks.sol";` imports unused alias `IHooks` in `Lock.sol`.
- The import `import {IHooks} from "../interfaces/IHooks.sol";` imports unused alias `IHooks` in `NonZeroDeltaCount.sol`.
- The import `import {BalanceDelta, BalanceDeltaLibrary, toBalanceDelta} from "./types/BalanceDelta.sol";` imports unused alias `toBalanceDelta` in `PoolManager.sol`.
- The import `import {Currency} from "../types/Currency.sol";` imports unused alias `Currency` in `StateLibrary.sol`.
- The import `import {PoolId} from "../types/PoolId.sol";` imports unused alias `PoolId` in `TransientStateLibrary.sol`.
- The import `import {Position} from "./Position.sol";` imports unused alias `Position` in `TransientStateLibrary.sol`.

Consider removing unused imports to improve the overall clarity and readability of the codebase.

***Update:** Resolved in pull request #763.*

N-09 Unused Error

The `TickNotInitialized` error in `Pool.sol` is defined but never used.

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

Update: *Resolved in pull request #764.*

N-10 Magic Numbers

Throughout the codebase, there are a few instances where literal values are used directly for arithmetic operations:

- The `255738958999603826347141` literal number in `TickMath.sol`
- The `3402992956809132418596140100660247210` literal number in `TickMath.sol`
- The `291339464771989622907027621153398088495` literal number in `TickMath.sol`

In order to improve code readability, consider using a constant to define such values and document its purpose.

Update: *Acknowledged, not resolved.*

N-11 State Variable Visibility Not Explicitly Declared

Throughout the codebase, there are state variables that lack an explicitly declared visibility:

- The `IS_UNLOCKED_SLOT` state variable in `Lock.sol`
- The `NONZERO_DELTA_COUNT_SLOT` state variable in `NonZeroDeltaCount.sol`
- The `RESERVES_OF_SLOT` state variable in `Reserves.sol`

For clarity, consider always explicitly declaring the visibility of variables, even when the default visibility matches the intended visibility.

Update: *Resolved in pull request #766.*

N-12 Missing Named Parameters in Mappings

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType KeyName? => ValueType ValueName?)`. This updated syntax provides a more transparent representation of a mapping's purpose.

Within `ERC6909.sol`, there are multiple mappings without named parameters:

- The `isOperator` state variable
- The `balanceOf` state variable
- The `allowance` state variable

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

Update: *Resolved in pull request #767.*

N-13 Lack of Indexed Event Parameter

Within `IProtocolFees.sol`, the `ProtocolFeeControllerUpdated` event does not have an indexed parameter.

To improve the ability of off-chain services to search and filter for specific events, consider indexing event parameters.

Update: *Resolved in pull request #768.*

N-14 Missing Docstrings

Throughout the codebase, there are multiple code instances that do not have docstrings:

- The `OperatorSet`, `Approval` and `Transfer` events, the `isOperator`, `balanceOf` and `allowance` state variables, and the `transfer`, `transferFrom`, `approve`, `setOperator` and `supportsInterface` functions in `ERC6909.sol`
- The `IERC6909Claims` interface in `IERC6909Claims.sol`
- The `IExtsload` interface and the `IExttload` interface in `IExtsload.sol`
- The `IPoolManager` interface in `IPoolManager.sol`
- The `IProtocolFeeController` interface in `IProtocolFeeController.sol`

- The `IProtocolFees` interface, the `ProtocolFeeControllerUpdated`, `ProtocolFeeUpdated`, and the `protocolFeeController` functions in `IProtocolFees.sol`
- The `IUnlockCallback` interface in `IUnlockCallback.sol`
- The `LPFeeLibrary` library and the `DYNAMIC_FEE_FLAG`, `OVERRIDE_FEE_FLAG`, `REMOVE_OVERRIDE_MASK`, and `MAX_LP_FEE` state variables in `LPFeeLibrary.sol`
- The `Pool` library in `Pool.sol`
- The `ProtocolFeeLibrary` library and the `MAX_PROTOCOL_FEE` state variable in `ProtocolFeeLibrary.sol`
- The `ProtocolFees` abstract contract and the `protocolFeesAccrued` and `protocolFeeController` state variables in `ProtocolFees.sol`
- The `Reserves` library in `Reserves.sol`
- The `Slot0Library` library in `Slot0.sol`
- The `StateLibrary` library and the `POOLS_SLOT`, `FEE_GROWTH_GLOBAL0_OFFSET`, `FEE_GROWTH_GLOBAL1_OFFSET`, `LIQUIDITY_OFFSET`, `TICKS_OFFSET`, `TICK_BITMAP_OFFSET` and `POSITIONS_OFFSET` state variables in `StateLibrary.sol`
- The `TransientStateLibrary` library and the `NONZERO_DELTA_COUNT_SLOT` and `IS_UNLOCKED_SLOT` state variables in `TransientStateLibrary.sol`
- The `BalanceDeltaLibrary` library and the `ZERO_DELTA` state variable in `BalanceDelta.sol`
- The `BeforeSwapDeltaLibrary` library and the `ZERO_DELTA` state variable in `BeforeSwapDelta.sol`
- The `NATIVE` state variable in `Currency.sol`

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:** Partially resolved in pull request #773. Only the first item of the list has not been addressed.*

N-15 Incomplete Docstrings

Throughout the codebase, there are several instances of incomplete docstrings:

- The return value is not documented for the `transfer`, `transferFrom`, `approve`, `setOperator` functions in `IERC6909Claims.sol`.

- The `delta` parameter is not documented for the `afterAddLiquidity` and `afterRemoveLiquidity` functions in `IHooks.sol`.
- The return value is not documented for the `MAX_TICK_SPACING` and `MIN_TICK_SPACING` functions in `IPoolManager.sol`.
- The `currency` parameter and the return value are not documented for the `sync` function in `IPoolManager.sol`.
- The `key`, `sqrtPriceX96`, and `hookData` parameters and the return value are not documented for the `initialize` function in `IPoolManager.sol`.
- The `key`, `amount0`, `amount1`, and `hookData` parameters and the return value are not documented for the `donate` function in `IPoolManager.sol`.
- The `currency`, `to`, and `amount` parameters are not documented for the `take` function in `IPoolManager.sol`.
- The `to`, `id`, and `amount` parameters are not documented for the `mint` function in `IPoolManager.sol`.
- The `from`, `id`, and `amount` parameters are not documented for the `burn` function in `IPoolManager.sol`.
- The `token` parameter and the return value are not documented for the `settle` function in `IPoolManager.sol`.
- The `key` and `newDynamicLPFee` parameters are not documented for the `updateDynamicLPFee` function in `IPoolManager.sol`.
- The return value is not documented for the `protocolFeeForPool` function in `IProtocolFeeController.sol`.
- The parameter and the return value are not documented for the `protocolFeesAccrued` function in `IProtocolFees.sol`.
- The `key` and the second parameter are not documented for the `setProtocolFee` function in `IProtocolFees.sol`.
- The parameter for the `setProtocolFeeController` function in `IProtocolFees.sol` is not documented.
- The parameters and the return value are not documented for the `collectProtocolFees` function in `IProtocolFees.sol`.
- The `fee` parameter is not documented for the `isValidHookAddress` function in `Hooks.sol`.
- The return values for all functions in `LPFeeLibrary.sol` are 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).

Update: *Resolved in pull request #775.*

N-16 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, all contracts do not have a security contact specified.

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:** Resolved in pull request #774. The team decided to create a `SECURITY.md` markdown file in the root of the project repository with the necessary security contacts.*

Client Reported

CR-01 License Limitation Can Be Skipped

The purpose of the `noDelegateCall` modifier in the `unlock` function is to prevent proxy contracts from making delegate calls to the Uniswap v4 core contracts. This restriction is mainly due to license limitations. However, during the audit, the team was notified that this limitation can be bypassed by creating a custom contract that implements the same `unlock` function and writes to the same slot for the lock value, thereby allowing delegate calls to all other external functions, effectively circumventing the restriction.

The Uniswap team might consider adding the `noDelegateCall` modifier to all external functions, rather than only using it in the `unlock` function.

***Update:** Resolved in pull request #743. The team added the `noDelegateCall` modifier to all relevant external functions.*

Conclusion

Uniswap v4 is an AMM that uses the concentrated liquidity model by default and can customize the model along with other parts of the protocol through the use of hooks. It also introduces singleton architecture, flash accounting, dynamic liquidity providers fee, donation to liquidity providers, and native token support.

One critical- and several medium-severity vulnerabilities were discovered along with some other issues of lower severity. In addition, suggestions were also made to improve the readability and clarity of the codebase in order to facilitate future audits and development. The codebase was found to be robust and generally well-documented. However, extensive use of assembly and other optimization techniques has significantly increased code complexity, thereby leaving room for undiscovered issues.

The Uniswap team was exceptionally responsive and provided us with extensive documentation about the project.