cecuro

Audit Report

March 11, 2026

PROJECT

Uniswap

Audit Overview

Project:	Uniswap
Repository:	https://github.com/Uniswap/v4-core
Audit Date:	March 11, 2026
Commit:	b619b671
Scope:	45 files

0	0	1	30	39
CRITICAL	HIGH	MEDIUM	LOW	INFO

Audit Scope

The following 45 files were included in this security audit:

- src/ERC6909.sol
- src/ERC6909Claims.sol
- src/Extsload.sol
- src/Exttload.sol
- src/NoDelegateCall.sol
- src/PoolManager.sol
- src/ProtocolFees.sol
- src/libraries/BitMath.sol
- src/libraries/CurrencyDelta.sol
- src/libraries/CurrencyReserves.sol
- src/libraries/CustomRevert.sol
- src/libraries/FixedPoint128.sol
- src/libraries/FixedPoint96.sol
- src/libraries/FullMath.sol
- src/libraries/Hooks.sol
- src/libraries/LPFeeLibrary.sol
- src/libraries/LiquidityMath.sol
- src/libraries/Lock.sol
- src/libraries/NonzeroDeltaCount.sol
- src/libraries/ParseBytes.sol
- src/libraries/Pool.sol
- src/libraries/Position.sol
- src/libraries/ProtocolFeeLibrary.sol
- src/libraries/SafeCast.sol
- src/libraries/SqrtPriceMath.sol
- src/libraries/StateLibrary.sol
- src/libraries/SwapMath.sol
- src/libraries/TickBitmap.sol
- src/libraries/TickMath.sol
- src/libraries/TransientStateLibrary.sol
- src/libraries/UnsafeMath.sol
- src/types/BalanceDelta.sol
- src/types/BeforeSwapDelta.sol
- src/types/Currency.sol
- src/types/PoolId.sol
- src/types/PoolKey.sol
- src/types/Slot0.sol
- src/interfaces/IExtsload.sol
- src/interfaces/IExttload.sol
- src/interfaces/IHooks.sol
- src/interfaces/IPoolManager.sol
- src/interfaces/IProtocolFees.sol
- src/interfaces/callback/IUnlockCallback.sol
- src/interfaces/external/IERC20Minimal.sol
- src/interfaces/external/IERC6909Claims.sol

Findings

MEDIUM

ERC6909 approve() has classic nonzero->nonzero allowance race (double-spend)

Location:

`src/ERC6909.sol:35-56`

Description:

`ERC6909.approve(spender,id,amount)` overwrites an existing allowance without requiring it to be set to 0 first, enabling the well-known ERC20-style allowance front-running race.

A spender can front-run an allowance change and spend both the old and new allowance.

Vulnerable Code:

```
function transferFrom(address sender, address receiver, uint256 id, uint256 amount) public
virtual returns (bool) {
 if (msg.sender != sender && !isOperator[sender][msg.sender]) {
 uint256 allowed = allowance[sender][msg.sender][id];
 if (allowed != type(uint256).max) allowance[sender][msg.sender][id] = allowed - amount;
 }
 ...
}

function approve(address spender, uint256 id, uint256 amount) public virtual returns (bool) {
 allowance[msg.sender][spender][id] = amount;
 emit Approval(msg.sender, spender, id, amount);
 return true;
}
```

- `src/ERC6909.sol:35-56`

Impact:

A spender can extract more ERC6909 claim tokens than the owner intended.

Recommendation:

Provide safer allowance-changing patterns (e.g., increase/decrease allowance) and/or require setting allowance to 0 before nonzero updates.

LOW

ERC20 transfer can be silently uncredited if `settle()` is called without prior `sync(currency)`

Locations:

- `src/PoolManager.sol:278-287`
- `src/PoolManager.sol:347-364`
- `src/libraries/CurrencyReserves.sol:21-31`
- `src/interfaces/IPoolManager.sol:180-186`
- `test/Sync.t.sol:205-252`

`PoolManager._settle()` determines settlement mode from transient `CurrencyReserves.CURRENCY_SLOT`. When that slot is zero, `_settle()` always executes the native path (`paid = msg.value`) and does not read ERC20 balance deltas.

Because zero is also the default unsynced state (and is written by `resetCurrency()`), an ERC20 transfer sent to `PoolManager` before `sync(currency)` is not credited by `settle()`:

```
Currency currency = CurrencyReserves.getSyncedCurrency();
if (currency.isAddressZero()) {
 paid = msg.value; // ignores ERC20 balance changes
} else {
 uint256 reservesBefore = CurrencyReserves.getSyncedReserves();
 uint256 reservesNow = currency.balanceOfSelf();
 paid = reservesNow - reservesBefore;
 CurrencyReserves.resetCurrency();
}
_accountDelta(currency, paid.toInt128(), recipient);
```

Concrete trace (amount=10):

1. `take(currency0, ..., 10)` opens router delta `-10`.
2. User transfers 10 `currency0` directly to manager **without sync**.
3. `settle()` sees synced currency = `address(0)` and credits `paid=0` (native), so currency0 debt stays `-10`.
4. Router must send another 10 after a proper `sync(currency0)` to close debt.
5. Net effect: manager ends +10 currency0, user ends -10 currency0 (as shown in `test_settle_nonNative_withoutSync_loseFunds`).

There is no direct in-core path to retro-credit that first mistaken transfer to the payer (`settleFor`, `take`, `clear`, `mint/burn`, and `collectProtocolFees` do not recover prior unaccounted ERC20 inflows).

Impact framing:

This is a real accounting footgun that can cause permanent loss for the payer under incorrect integration order. However, it is an integration misuse scenario explicitly documented in `IPoolManager.sync` and helper libraries already enforce the correct sequence. It is therefore best classified as **Low** severity, not a permissionless exploit against correct integrations.

Recommendation:

- Add stronger guardrails to fail fast on unsynced ERC20 settle flows (e.g., explicit synced-state marker distinct from native, or optional strict mode that reverts on `settle()` with `syncedCurrency==0 && msg.value==0`).
- Keep/expand integration guidance requiring `sync(currency) -> transfer -> settle` for ERC20.

LOW

Pool initialization accepts no-code hook addresses, allowing flag-gated actions to revert and potentially lock LP positions

Locations:

`src/libraries/Hooks.sol:108-126`

`src/PoolManager.sol:117-142`

`src/libraries/Hooks.sol:150-153`

`src/libraries/Hooks.sol:193-244`

`src/PoolManager.sol:145-183`

`PoolManager.initialize` only validates hook *address bits* via `Hooks.isValidHookAddress` and does not verify that a nonzero hook address has deployed code.

As a result, a pool can be initialized with a no-code hook address that has valid permission bits (e.g. `0x...0100` for `AFTER_REMOVE_LIQUIDITY_FLAG`) and fixed fee `3000`:

- `isValidHookAddress` returns `true` because `(uint160(address(self)) & ALL_HOOK_MASK) > 0`.
- No `code.length` / `extcodesize` check is performed in this path.

Later, when a flagged hook is invoked, `callHook` performs a low-level `call` and validates returned selector:

```
if (result.length < 32 || result.parseSelector() != data.parseSelector()) {
 InvalidHookResponse.selector.revertWith();
}
```

A call to a no-code address returns empty returndata, so this reverts with `InvalidHookResponse`.

Concrete scenario:

1. Initialize pool with hooks=`0x...0100` (no initialize bits set) ' initialization succeeds.
2. Add liquidity (`liquidityDelta > 0`) with no add-hook bits set ' succeeds.
3. Remove liquidity (`liquidityDelta < 0`) with remove-hook bit set ' remove path calls hook and reverts `InvalidHookResponse`.

Because the remove transaction reverts atomically, attempted withdrawal changes do not persist, leaving liquidity effectively stuck for affected participants unless the hook address later becomes callable (e.g., counterfactual deployment). Scope is per-pool/per-flag, not protocol-wide.

Recommendation:

- During initialization, require `address(hooks).code.length > 0` for nonzero hooks addresses, **or**
- Explicitly support deferred/counterfactual hooks but gate with an additional safety flag/acknowledgement and clear integrator guidance.
- At minimum, document that no-code hooks with active flags can brick corresponding actions.

LOW

Unbounded hook/transfer returndata copying enables gas-amplified reverts for malicious hooks or tokens

Locations:

`src/libraries/Hooks.sol:130-147`

`src/libraries/CustomRevert.sol:83-117`

`src/types/Currency.sol:51-53`

`src/types/Currency.sol:82-85`

`Hooks.callHook` and `CustomRevert.bubbleUpAndRevertWith` copy `returndatasize()` bytes without an upper bound.

- In `Hooks.callHook`, after a successful external call, the library allocates `result` using full `returndatasize()` and executes `returndatacopy` before response validation.
- On failed hook/token/native-transfer calls, `bubbleUpAndRevertWith` re-encodes the *entire* revert payload into `WrappedError` via unbounded `returndatacopy`.

This allows a malicious hook or malicious/non-standard token to return/revert with very large payloads and force high caller-side memory expansion/copy cost, potentially causing caller OOG before normal error handling. Example gas model: ~512KB returndata costs ~623k gas just for copy+memory expansion, and ~1MB costs ~2.29M; with EIP-150 retained gas (~1/64), this can exceed caller-retained gas when callee also consumes forwarded gas.

Reachable paths include hook-enabled `initialize/modifyLiquidity/swap/donate` (via `Hooks.callHook`) and payout paths (`PoolManager.take` / `ProtocolFees.collectProtocolFees`) via `CurrencyLibrary.transfer` -> `bubbleUpAndRevertWith`.

Impact is primarily **gas griefing / local DoS** of operations that interact with those malicious components. It does **not** create protocol-wide persistent lockout: transactions revert atomically, and lock/delta transient state rolls back.

Recommendation

- Bound copied returndata/revertdata (e.g., cap at a fixed maximum and truncate).
- Consider emitting/encoding only bounded prefixes in wrapped errors.
- Optionally limit gas forwarded to untrusted hook calls if compatible with design.

LOW

Swap gas scales linearly with crossed initialized ticks, enabling high-crossing swap DoS scenarios

Locations:

`src/libraries/Pool.sol:340-434`

`src/libraries/TickBitmap.sol:85-121`

`src/libraries/Pool.sol:143-177`

`src/PoolManager.sol:117-121`

`Pool.swap()` iterates in a `while` loop until amount exhaustion or price limit. There is no explicit iteration cap; each iteration advances to the next initialized tick (or a word boundary) via `nextInitializedTickWithinOneWord()`.

Because `TickBitmap.nextInitializedTickWithinOneWord()` only resolves within one 256-bit word and returns the nearest initialized tick or boundary, runtime scales with traversal steps (initialized crossings + empty-word traversals), not with a fixed upper gas budget.

A permissionless actor can increase future swap cost by adding many minimal adjacent ranges (e.g., at `tickSpacing=1`, ranges `[-1,0],[-2,-1],...`). This densifies `tickBitmap`/`ticks`, causing many `initialized=true` crossings. On each initialized crossing, `crossTick()` mutates per-tick storage (`feeGrowthOutside*`), adding substantial per-crossing gas.

Concrete path simulation:

- Pool initialized at tick 0, spacing 1.
- Add N tiny adjacent positions with `liquidityDelta=1`.
- Execute `swap(zeroForOne=true, amountSpecified=-1e18, sqrtPriceLimitX96=MIN_SQRT_PRICE+1)`.
- Swap performs O(N) initialized crossings through the bombed region before continuing toward limit.

Analytical gas lower bound from storage writes alone is roughly linear (`~20k * (N-1)` for one `0->nonzero` fee-growth-outside slot per crossed initialized tick once fee growth becomes nonzero), so low-thousands of initialized crossings can approach/exceed typical L1 block gas budgets for a single transaction.

Impact is best framed as **high-crossing swap executability degradation** (especially for wide-limit/exact-output or multi-hop routes), not universal permanent pool DoS: users can often split trades or use tighter limits.

LOW

NatSpec/Event docs describe pool deltas, but `BalanceDelta` values are caller-perspective

Locations:

```
src/libraries/Pool.sol:141
src/libraries/Pool.sol:274
src/interfaces/IPoolManager.sol:84-85
src/PoolManager.sol:240-244
src/PoolManager.sol:225
src/libraries/Pool.sol:450-457
```

`BalanceDelta` semantics in execution are caller-centric, but several comments describe them as pool-balance deltas.

What code does (caller perspective):

- In `Pool.swap`, for `zeroForOne=true` and exact-input (`amountSpecified < 0`), final delta assignment uses:
  - `amount0 = amountSpecified - amountSpecifiedRemaining` (negative when caller pays token0)
  - `amount1 = amountCalculated` (positive when caller receives token1)

(`src/libraries/Pool.sol:450-457`)

- `PoolManager._swap` emits `Swap(..., delta.amount0(), delta.amount1(), ...)` directly (`src/PoolManager.sol:240-244`).
- The same delta is then booked to `msg.sender` via `_accountPoolBalanceDelta(key, swapDelta, msg.sender)` (`src/PoolManager.sol:225`), which only makes sense as caller deltas.

So signs are: **negative = caller owes PoolManager**, **positive = caller is owed/claimable**.

Where docs are inconsistent:

- `Pool.modifyLiquidity` NatSpec says return is “deltas of the token balances of the pool” (`src/libraries/Pool.sol:141`).
- `Pool.swap` NatSpec says it returns “amount deltas of the pool” (`src/libraries/Pool.sol:274`).

- `IPoolManager.Swap` event NatSpec says `amount0/amount1` are pool balance deltas (`src/interfaces/IPoolManager.sol:84-85`).

These statements conflict with actual emitted/accounted values.

Impact:

No direct on-chain exploit path, but integrators/indexers relying on these comments can invert sign interpretation and mis-attribute flows. This is primarily a documentation/interface semantic risk.

Recommendation:

Standardize NatSpec to caller-perspective wording for all `BalanceDelta` values and swap event params (e.g., negative = caller owes, positive = caller receives/claimable). If pool-perspective event values are preferred, emit negated values consistently and update accounting/docs accordingly.

LOW

PoolManager is intentionally non-proxy-safe: delegatecall proxy deployments break core entrypoints and leave ProtocolFees ownership uninitialized

Locations:

`src/NoDelegateCall.sol:14-31`

`src/PoolManager.sol:101`

`src/PoolManager.sol:117`

`src/PoolManager.sol:145-149`

`src/PoolManager.sol:186-190`

`src/PoolManager.sol:255-259`

`src/ProtocolFees.sol:26`

`lib/solmate/src/auth/Owned.sol:17-33`

`PoolManager`/`ProtocolFees` use constructor-based initialization and `NoDelegateCall` immutables, so delegatecall-proxy deployments (Transparent/UUPS/clone-style) are incompatible.

What happens:

1. `NoDelegateCall` stores immutable `original = address(this)` in constructor (`NoDelegateCall.sol:16-20`).
2. When `PoolManager` logic is executed via proxy delegatecall, runtime `address(this)` is the proxy, while `original` remains the implementation address.
3. Therefore `checkNotDelegateCall()` (`NoDelegateCall.sol:24-26`) fails and `DelegateCallNotAllowed` is raised for guarded functions.
4. In `PoolManager`, core entrypoints `initialize`, `modifyLiquidity`, `swap`, and `donate` are guarded by `noDelegateCall`, so proxied calls revert before pool/accounting mutation.

Additionally, `ProtocolFees` ownership is constructor-only (`ProtocolFees.sol:26` via `Owned`). In delegatecall proxy deployments, that constructor does not initialize proxy storage, so `owner` in proxy storage remains unset (typically `address(0)`), making `onlyOwner` paths such as `setProtocolFeeController` unusable unless a separate initializer is added externally.

Impact:

For a deployment that incorrectly places `PoolManager` behind a delegatecall proxy, core pool actions are effectively non-functional (revert) and protocol fee admin setup can be stuck. This is a deployment/integration hazard rather than a permissionless exploit against canonical singleton deployments.

Recommendation:

Explicitly document that `PoolManager` must be deployed directly (non-proxy, non-clone delegatecall). If upgradeable/proxy support is ever desired, redesign with initializer-based ownership/state setup and a proxy-safe alternative to `NoDelegate-Call` semantics.

LOW

`CurrencyLibrary.transfer` treats no-code ERC20 targets as success, allowing silent no-op payouts

Locations:

```
src/types/Currency.sol:64-74
```

```
src/ProtocolFees.sol:54-56
```

```
src/PoolManager.sol:293-294
```

`CurrencyLibrary.transfer`'s ERC20 path accepts empty return data as success and does not verify that `currency` has contract code.

In assembly, success is computed as:

- low-level `call(...)` result is true, and
- `(returndatasize == 0) || (returndata is 32+ bytes and first word == 1)`

For an EOA / destroyed-contract address, `call` succeeds with `returndatasize == 0`, so the transfer is treated as successful even though no token state can change.

This behavior propagates to state-changing call sites that update accounting before transfer:

- `collectProtocolFees` decreases `protocolFeesAccrued[currency]` then calls `currency.transfer(...)`.
- `take` debits caller delta then calls `currency.transfer(...)`.

So accounting can be consumed while no payout occurs.

Impact calibration:

This is a real behavior, but impact depends on malformed/non-standard asset selection (e.g., EOA address, or token behavior that no-ops on transfer while returning empty/success). Standard ERC20s are not affected. `collectProtocolFees` is also controller-gated. Therefore severity is best set to **Low** (robustness / compatibility foot-gun with conditional accounting drift), not higher.

Recommendation:
If stricter semantics are desired, add code-existence validation in the ERC20 path (e.g., `address(currency).code.length > 0`) and/or enforce token validation at pool creation/integration boundaries. Keep in mind this may reduce compatibility with intentionally non-standard tokens that return no data.

LOW

Dynamic-fee pools accept nonzero hook addresses with no permission flags (configuration footgun)

Locations:

- src/libraries/Hooks.sol:108-126
- src/PoolManager.sol:126
- src/libraries/Hooks.sol:177-204
- src/libraries/Hooks.sol:247-333

`Hooks.isValidHookAddress` explicitly permits a nonzero hook address with **zero hook flag bits** when `fee` is dynamic (`0x800000`):

```
return address(self) == address(0)
 ? !fee.isDynamicFee()
 : (uint160(address(self)) & ALL_HOOK_MASK > 0 || fee.isDynamicFee());
```

`PoolManager.initialize` relies on this check:

```
if (!key.hooks.isValidHookAddress(key.fee)) revert HookAddressNotValid(...);
```

So a pool with `key.fee = DYNAMIC_FEE_FLAG` and `key.hooks = 0x1000...000` (no low 14 bits set) initializes successfully. After that, hook callbacks are never executed because each callback site is gated by `hasPermission(...)` checks (before/after initialize, modifyLiquidity, swap, donate), and all permission checks are false.

Impact: this is a **silent configuration risk**. Integrators may set a nonzero hook address expecting policy logic to run, but callbacks remain disabled. In dynamic-fee pools this can also leave behavior different from expectation (e.g., no callback-based fee override/business rules). This is primarily deployer/integrator misconfiguration rather than a permissionless theft vector.

Recommendation: keep current flexibility if desired, but add an explicit guardrail for dynamic-fee pools (e.g., require at least one callback flag, or explicit opt-in for no-flag dynamic pools via separate sentinel/parameter/event) to reduce accidental misconfiguration.

LOW

Dynamic-fee hooks can set 100% per-swap fee, and core swap API has no caller-side max-fee bound

Locations:

```
src/interfaces/IPoolManager.sol:143-150
src/PoolManager.sol:186-226
src/libraries/Hooks.sol:246-263
src/libraries/LPFeeLibrary.sol:24-26
src/libraries/Pool.sol:297-313
src/libraries/SwapMath.sol:63-86
```

`PoolManager.swap` does not accept a user-provided max-fee parameter; `SwapParams` only includes direction, `amountSpecified`, and `sqrtPriceLimitX96`.

For dynamic-fee pools, the configured hook can provide a per-swap LP fee override in `beforeSwap` (`Hooks.beforeSwap`), and that override is accepted up to `MAX_LP_FEE = 1_000_000` (100%) after flag removal/validation.

When effective `swapFee == 1_000_000`, `Pool.swap` allows exact-input swaps (it only blocks exact-output at this boundary). In `SwapMath.computeSwapStep`, exact-input with 100% fee yields `amountRemainingLessFee = 0`; the step consumes the full specified input as `feeAmount` and produces `amountOut = 0` (no price movement). Thus a swap can settle as full input paid, zero output received.

This is a real behavior and creates a honeypot/sharp-edge risk for integrations trading against untrusted dynamic hooks.

Impact: A swapper can lose their full input amount on a single exact-input swap (all taken as fees) if the integration does not enforce output constraints externally.

Why Low severity: This is largely a trust/integration risk (pool hook has intended fee control for dynamic-fee pools), and robust routers can atomically prevent loss via min-out / post-swap-delta checks and revert otherwise.

Recommendation:

1. Add an optional caller-provided max-fee (or max total fee) bound in core/periphery swap interfaces, or

2. Strictly document that integrations must enforce min-out / returned `swapDelta` checks when interacting with hooks/dynamic-fee pools, and

3. Prefer hook allowlists for routing.

LOW

Settlement trusts ERC20-reported `balanceOf` delta, allowing dishonest tokens to clear debt without transfer

Locations:

`src/PoolManager.sol:277-286`

`src/PoolManager.sol:348-364`

`src/types/Currency.sol:90-95`

`src/libraries/CurrencyReserves.sol:27-37`

`PoolManager` derives ERC20 settlement credit from two external `balanceOf(address(this))` reads (`sync` then `_settle`) rather than from transfer success/accounted inflow.

- `sync(currency)` snapshots `balanceOfSelf()` into transient reserves.
- `_settle(recipient)` computes `paid = reservesNow - reservesBefore` from another `balanceOfSelf()` call and credits `_accountDelta(currency, paid, recipient)`.
- For ERC20, `balanceOfSelf()` is an external call to `IERC20Minimal.balanceOf`.

Because this accounting trusts token-reported balances, a malicious ERC20 can over-report `balanceOf(PoolManager)` at settle time and create arbitrary positive `paid` without transferring tokens.

Concrete path in one unlock callback:

1. Swap in pool `(maliciousToken, honestToken)` to create attacker deltas `malicious < 0`, `honest > 0`.
2. Call `sync(maliciousToken)`.
3. Call `settle()`; malicious token reports inflated balance so fake `paid` clears malicious debt.
4. Call `take(honestToken, ...)` to withdraw real honest tokens.
5. End with zero deltas (using exact fake amount, repeated settle, or `clear` for overpayment), so unlock succeeds.

This can drain counterpart assets in pools that include dishonest tokens.

Severity is **Low** because exploitability depends on malicious/broken asset behavior in a permissionless listing environment (asset-trust assumption), not a protocol-wide break for honest ERC20s.

Recommendation: explicitly document this trust assumption, and for production deployments consider token allowlists / listing controls where this risk is unacceptable.

LOW

PoolManager intentionally skips hook callbacks on hook self-calls, allowing callback-only policies to be bypassed via hook forwarding entry-points

Locations:

`src/libraries/Hooks.sol:169-174`

`src/libraries/Hooks.sol:216-217`

`src/libraries/Hooks.sol:252-253`

`src/libraries/Hooks.sol:292-293`

`src/PoolManager.sol:201-201`

`src/PoolManager.sol:220-220`

`src/interfaces/IHooks.sol:14`

`Hooks` suppresses callback execution whenever the PoolManager caller is the hook contract itself (`msg.sender == address(self)`).

- `noSelfCall` (`Hooks.sol:169-174`) skips bodies for `beforeInitialize`, `afterInitialize`, `beforeModifyLiquidity`, `beforeDonate`, `afterDonate`.
- `afterModifyLiquidity`, `beforeSwap`, and `afterSwap` also short-circuit with explicit early returns (`216`, `252`, `292`), returning unchanged/default outputs.

This creates a real bypass condition for hook authors who enforce restrictions only in callbacks.

Concrete trace:

If a hook has:

1. restrictive `beforeSwap` policy (e.g., `require(sender == admin)`), and
2. a permissionless external function that forwards into `PoolManager.unlock(...)` and then `PoolManager.swap(...)` from `unlockCallback`,

then an unprivileged user can call that external function, causing `PoolManager.swap` to execute with `msg.sender == hook`. `Hooks.beforeSwap` returns early and never calls `IHooks.beforeSwap`, so the restrictive callback check is not executed.

Impact:

This is primarily a hook-integration footgun / semantic mismatch:

- callback-based hook policies (ACL, custom invariants, extra fees) can be bypassed if the hook exposes permissive forwarding entrypoints and does not duplicate checks there.
- core PoolManager accounting/settlement is still enforced (`unlock` requires `NonzeroDeltaCount == 0`), so this is not by itself a direct core-funds theft primitive.

Recommendation:

Document self-call suppression prominently for hook developers (including in `IHooks` docs), and require hook authors to enforce policy in their own external entrypoints/unlock callback paths, not only inside PoolManager-triggered hook callbacks.

LOW

Initialize hooks can open `unlock` sessions while initialization is in progress

Locations:

`src/PoolManager.sol:104-113`

`src/PoolManager.sol:117-142`

`src/libraries/Hooks.sol:176-191`

`PoolManager.initialize()` is callable without `onlyWhenUnlocked` and invokes external hooks both before and after pool state initialization.

During a normal direct `initialize` call, the manager is in locked mode (`Lock.isUnlocked()==false`). Inside `beforeInitialize`/`afterInitialize`, the hook can call `PoolManager.unlock(...)` because `unlock` only rejects calls when already unlocked.

Concrete trace:

1. `initialize(keyB, 792281625142643375935439503360)` enters while locked.
2. `beforeInitialize` hook is called.
3. Hook calls `unlock(data)` -> lock flips to unlocked, callback executes unlocked-only actions (e.g., `swap` on another pool), then relocks if deltas are settled.
4. Outer `initialize` continues, initializes pool B.
5. `afterInitialize` hook can call `unlock(data)` again and execute unlocked-only actions (e.g., `modifyLiquidity` on the newly initialized pool).

So nested unlocked sessions during a single initialize are reachable and can persist state changes, provided each unlock callback settles to `NonzeroDeltaCount == 0`. If not settled, `CurrencyNotSettled` reverts atomically.

Impact is primarily execution-model complexity: integrations that assume "manager locked during initialize implies no unlocked-only state transitions can occur" are incorrect. This increases reentrancy/control-flow surface during initialization, but does not by itself produce direct fund theft because accounting checks still apply and unsettled deltas revert.

Recommendation: explicitly document this behavior as part of the hook model (initialize hooks may open unlock sessions), or add a guard if the intended model is to disallow unlocked actions during initialize hooks.

LOW

Hook-returned deltas can fully bypass core swap execution and protocol-fee accrual

Locations:

`src/libraries/Hooks.sol:251-278`

`src/libraries/Pool.sol:315-318`

`src/PoolManager.sol:233-238`

`src/libraries/Hooks.sol:294-312`

`src/PoolManager.sol:223-225`

`beforeSwap` hooks with `BEFORE_SWAP_RETURNS_DELTA_FLAG` can set `amountToSwap` to zero by returning a specified delta equal to `-params.amountSpecified`.

```
amountToSwap = params.amountSpecified;
...
amountToSwap += hookDeltaSpecified;
// only exactIn/exactOut sign is enforced
```

(`Hooks.beforeSwap`, `src/libraries/Hooks.sol:251-278`)

If `amountToSwap == 0`, the core pool swap returns early with no protocol accrual:

```
if (params.amountSpecified == 0) return (ZERO_DELTA, 0, swapFee, result);
```

(`Pool.swap`, `src/libraries/Pool.sol:315-318`)

And PoolManager only accrues protocol fees from `amountToProtocol` returned by core swap:

```
(BalanceDelta delta, uint256 amountToProtocol, ...) = pool.swap(params);
if (amountToProtocol > 0) _updateProtocolFees(inputCurrency, amountToProtocol);
```

(`src/PoolManager.sol:233-238`)

Despite core no-op, hook deltas are still applied to caller/hook balances in `afterSwap` and then accounted in PoolManager:

- `swapDelta = swapDelta - hookDelta` (`Hooks.sol:304-312`)

- `_accountPoolBalanceDelta(... hookDelta, hook)` and `_accountPoolBalanceDelta(... swapDelta, caller)` (`PoolManager.sol:223-225`)

So a hook can implement swap economics via custom accounting while protocol fees remain zero.

Impact:

This is a real protocol-revenue bypass for hook-mediated swap volume in hook-enabled pools: repeated swaps can transfer value between caller and hook without increasing `protocolFeesAccrued`.

Severity rationale:

Low: behavior is real and reachable, but it does not directly steal user funds or break core access control; it mainly affects protocol monetization assumptions in pools that opt into such hooks.

Recommendation:

If protocol fees are intended to apply to all swap-like value transfer, enforce policy at the hook boundary (e.g., disallow full zeroing of nonzero user `amountSpecified`, or explicitly levy protocol fee on hook-returned swap deltas). Otherwise, document this as an intentional capability of custom-accounting hooks so fee expectations are explicit.

LOW

Protocol uses an implicit int128 numeric domain behind wider ABI types, causing boundary reverts and rare fee-accrual liveness lock

Locations:

- `src/types/BalanceDelta.sol:6-18`
- `src/libraries/SafeCast.sol:40-43`
- `src/libraries/SafeCast.sol:56-59`
- `src/PoolManager.sol:163`
- `src/libraries/Pool.sol:189`
- `src/libraries/Pool.sol:451-457`
- `src/libraries/Pool.sol:468`
- `src/libraries/Position.sol:93-96`

`v4-core` exposes several `int256/uint256` inputs at the interface layer, but core accounting is ultimately packed into `BalanceDelta` (`int128,int128`). This creates an effective runtime domain of `[-2^127, 2^127-1]` / `[0, 2^127-1]` for many paths.

What is happening:

- `BalanceDelta` packs two `int128` values (`src/types/BalanceDelta.sol:6-18`).
- `PoolManager.modifyLiquidity` downcasts external `int256 liquidityDelta` to `int128` (`src/PoolManager.sol:163`).
- `Pool.swap` computes in `int256` but downcasts final deltas to `int128` (`src/libraries/Pool.sol:451-457`).
- `Pool.donate` downcasts `uint256 amount{0,1}` to `int128` (`src/libraries/Pool.sol:468`).
- `Pool.modifyLiquidity` downcasts `Position.update` fee outputs (`uint256`) to `int128` (`src/libraries/Pool.sol:189`).
- `SafeCast` enforces these bounds and reverts at `x >= 2^127` (uint) or outside int128 range (`src/libraries/SafeCast.sol:40-43`, `:56-59`).

Concrete boundary behavior:

- `donate(amount0 = 2^127, amount1 = 0)` reverts with `SafeCastOverflow`.
- `modifyLiquidity(liquidityDelta = int256(2^127))` reverts with `SafeCastOverflow`.
- A deep-liquidity swap with `amountSpecified = -(2^127+1)` can execute swap math but revert at final `toInt128` packing in `Pool.swap`.

Aggregate/liveness edge case (real but low-likelihood):

Even if each individual donation/swap is within int128 limits, cumulative `feeGrowth` can make one later `modifyLiquidity` revert:

- Example: two donations of `2^126` each (both valid individually) to a position with liquidity 1 produce cumulative owed fees of `2^127`.
- On next `modifyLiquidity` (poke/remove), `feesOwed.toInt128()` reverts (`src/libraries/Pool.sol:189`).
- Because fee casting occurs before removal logic, both collect and exit paths for that position revert on the same path.

Impact:

- Immediate effect: ABI-valid large inputs can unexpectedly revert (integration/UX mismatch).
- Rare extreme effect: if a position accumulates `>= 2^127` uncollected fees in one token, that position's `modifyLiquidity` path can become unusable (practically permanent without impossible-scale wraparound).

Given required magnitudes, this is best classified as **Low** severity (real behavior, low practical likelihood in normal-value pools).

LOW

Zero-delta `modifyLiquidity` pokes are dispatched to remove-liquidity hooks (including return-delta path)

Locations:

`src/libraries/Hooks.sol:200-204`

`src/libraries/Hooks.sol:219-242`

`src/interfaces/IPoolManager.sol:132-137`

`src/PoolManager.sol:155-177`

`src/interfaces/IHooks.sol:64-93`

`IPoolManager.modifyLiquidity` documents that `liquidityDelta = 0` is a valid "poke" operation, but hook dispatch logic classifies this as the **remove-liquidity** path.

What happens in code:

- `PoolManager.modifyLiquidity` always calls hook routing before/after pool update (`src/PoolManager.sol:155`, `:177`).
- In `Hooks.beforeModifyLiquidity`, zero is routed to `beforeRemoveLiquidity` because the branch is `params.liquidityDelta <= 0` (`src/libraries/Hooks.sol:200-204`).
- In `Hooks.afterModifyLiquidity`, zero goes to the `else` branch and calls `afterRemoveLiquidity` (`src/libraries/Hooks.sol:219-242`).
- If `AFTER_REMOVE_LIQUIDITY_RETURNS_DELTA_FLAG` is enabled, the hook's returned delta is parsed and applied even for poke (`callerDelta = callerDelta - hookDelta`).

Concrete trace (poke):

For `liquidityDelta = 0`, core pool math returns no principal delta and only fee accrual delta. Then remove-hook callbacks run; if hook returns `hookDelta`, caller receives `feesAccrued - hookDelta`, and hook is credited `hookDelta`.

Security impact:

This is a **semantic/integration hazard** for hooked pools:

- Hooks that assume `afterRemoveLiquidity` only runs on negative deltas may unintentionally revert pokes.

- Poke-based fee realization/maintenance flows can fail in those pools.
- Remove-hook return-delta logic can charge/divert value during poke in ways integrators may not expect.

No core accounting invariant break was found; impact is pool-specific behavior/DoS surprise, so Low severity is appropriate.

Recommendation:

Handle `liquidityDelta == 0` explicitly (separate hook path or no add/remove hook), or clearly specify in interface/docs that pokes are treated as remove-liquidity hook calls so hook authors must branch on `params.liquidityDelta` accordingly.

LOW

Swaps can update `slot0` price/tick with zero token deltas when in-range liquidity is zero

Locations:

`src/libraries/Pool.sol:291-295`

`src/libraries/Pool.sol:340-379`

`src/libraries/Pool.sol:410-436`

`src/libraries/SwapMath.sol:63-87`

`src/libraries/SwapMath.sol:88-105`

`src/libraries/SqrtPriceMath.sol:204-210`

`src/libraries/SqrtPriceMath.sol:239-253`

`Pool.swap` does not prevent execution when `self.liquidity == 0`. In that state, `SwapMath.computeSwapStep` can still advance `sqrtPriceNextX96` to the step target while all token amounts remain zero.

Why this happens:

- `Pool.swap` initializes `result.liquidity = self.liquidity` and enters the while-loop as long as amount remains and price limit is not reached.
- With `liquidity == 0`, `SqrtPriceMath.getAmount0Delta/getAmount1Delta` return 0 (numerator includes liquidity), so in `computeSwapStep` both exact-in and exact-out branches can set `sqrtPriceNextX96 = sqrtPriceTargetX96` with `amountIn=0`, `amountOut=0`, `feeAmount=0`.
- Back in `Pool.swap`, `amountSpecifiedRemaining` is updated only by `amountIn/amountOut/feeAmount`, so it is unchanged while price can still move.
- At loop end, the new `result.tick` / `result.sqrtPriceX96` are committed to storage (`self.slot0 = ...`).

Concrete trace:

Initialized pool at tick 0 (`sqrtPriceX96 = Q96`), no LP liquidity, then call swap with nonzero amount and valid lower price limit:

1. Iteration 1 can set tick from 0 to -1 with zero amounts.
2. Next iteration moves price to `sqrtPriceLimitX96` with zero amounts.
3. Final `swapDelta` is `(0,0)`, but `slot0.tick/sqrtPriceX96` changed.

Impact:

No direct fund transfer occurs in this zero-liquidity path, but pool state (`slot0`) is permissionlessly movable at gas-only cost during zero-liquidity windows (e.g., right after initialize/before first LP, or when pool is out of range). This can mislead integrations/hooks or user workflows that treat `slot0` as meaningful without also checking liquidity.

Recommendation:

Either:

1. block price movement when in-range liquidity is zero (e.g., revert early in swap), or
2. explicitly document this behavior and require safe read patterns (`slot0` + liquidity checks) for oracle-like consumption.

LOW

`mint`/`burn` settle deltas are always applied to `msg.sender`, not `to`/`from` (underdocumented API semantics)

Locations:

`src/PoolManager.sol:321-327`

`src/PoolManager.sol:331-335`

`src/ERC6909Claims.sol:13-21`

`src/interfaces/IPoolManager.sol:211-225`

`PoolManager.mint` and `PoolManager.burn` intentionally decouple the ERC-6909 balance owner from the account whose pool delta is updated:

- `mint(to,id,amount)` does `_accountDelta(currency, -amount, msg.sender)` and `_mint(to,id,amount)`.
- `burn(from,id,amount)` does `_accountDelta(currency, +amount, msg.sender)` and `_burnFrom(from,id,amount)`.

So with concrete values (caller `C`, `to/from` = `T/F`, `amount=10`):

- `mint`: `currencyDelta(C)` decreases by 10; `balanceOf[T][id]` increases by 10; `currencyDelta(T)` unchanged.
- `burn`: `currencyDelta(C)` increases by 10; `balanceOf[F][id]` decreases by 10; `currencyDelta(F)` unchanged.

`_burnFrom` only enforces ownership/operator/allowance over the ERC-6909 balance; it does not redirect the delta to `from`. Therefore an approved spender can burn another account's claims and receive the positive delta themselves (then `take`/`mint` that value), up to granted allowance/operator rights.

This behavior is real and used by router patterns, but `IPoolManager` docs for `mint-/burn` do not clearly state caller-attributed delta semantics. The risk is integrator mis-accounting/unintended value routing due to API misinterpretation, not a direct protocol theft primitive.

LOW

`mint`/`burn` id canonicalization (uint160) is inconsistent with ERC6909 raw uint256 id keys, causing non-canonical-id balance/allowance mismatches

Locations:

```
src/PoolManager.sol:321-335
src/types/Currency.sol:114-118
src/ERC6909.sol:17-19
src/ERC6909.sol:25-43
src/ERC6909Claims.sol:13-21
```

`PoolManager.mint` and `PoolManager.burn` always canonicalize the supplied `uint256 id` through `CurrencyLibrary.fromId(id)` and then `currency.toId()`, which truncates to the lower 160 bits.

So for `idRaw` where upper 96 bits are non-zero, manager writes/reads ERC6909 state under `idCanon = uint160(idRaw)`, not `idRaw`.

Meanwhile ERC6909 `balanceOf`/`allowance`/`transfer`/`transferFrom` are keyed by the exact caller-provided `uint256 id` and do no canonicalization. This creates observable mismatch:

- After `manager.mint(to, idRaw, 100)`, `balanceOf(to, idRaw) == 0` but `balanceOf(to, idCanon) == 100`.
- `transfer(..., idRaw, 1)` reverts on underflow (zero balance at `idRaw`).
- `approve(spender, idRaw, amt)` stores allowance on the non-canonical key, which does not authorize actions that use canonical id (`idCanon`) such as manager burn paths.

This is a real integration footgun / namespace-collision issue for systems treating ERC6909 ids as opaque `uint256` values, although funds are not inherently lost (assets remain accessible via canonical id).

LOW

Unlock window is global and not bound to unlock initiator (caller-agnostic access while unlocked)

Locations:

`src/PoolManager.sol:96-114`

`src/libraries/Lock.sol:7-27`

`PoolManager`'s unlock gate is transaction-global: `onlyWhenUnlocked` checks only `Lock.isUnlocked()` and does not bind calls to the address that initiated `unlock()`.

What is happening:

- `unlock()` sets the global transient flag, calls `IUnlockCallback(msg.sender).unlockCallback(data)`, then requires global delta netting and relocks.
- During that callback, any contract that gains execution can call `onlyWhenUnlocked` functions (`swap`, `modifyLiquidity`, `donate`, `take`, `settle`, etc.), even if it is not the original unlock caller.
- `Lock` stores only one transient boolean slot; no `locker` address is tracked.

Concrete trace (from in-repo pattern):

`src/test/PoolNestedActionsTest.sol` demonstrates this exact cross-caller flow:

1. Contract A calls `manager.unlock(...)`.
2. Inside A's `unlockCallback`, A calls contract B (`NestedActionExecutor.execute`).
3. B calls `manager.swap(...)` successfully while manager is unlocked.

Security impact calibration:

- This behavior is real and can break integrator assumptions like "only my router executes manager actions during my unlock callback."
- However, in core it is mostly an **integration boundary** issue, not direct theft:
  - Deltas are keyed per target address; third-party calls primarily mutate their own deltas (except explicit APIs like `settleFor`).
  - If any participant leaves residual deltas, `unlock` reverts atomically (`CurrencyNotSettled`), so state/funds are not partially committed.

- Hooks are intentionally powerful and expected to call manager functions in this design.

Recommendation:
Document explicitly that unlocked access is global for the transaction (not caller-scoped). Periphery/integrators that need caller isolation should enforce it in their own callback logic (e.g., internal reentrancy/caller guards), or use a stricter manager wrapper that binds operations to the initiating locker when desired.

LOW

No built-in timelock or two-step governance for protocol fee controller actions

Locations:

`src/ProtocolFees.sol:29-32`

`src/ProtocolFees.sol:35-41`

`src/ProtocolFees.sol:44-57`

`lib/solmate/src/auth/Owned.sol:19-23`

`lib/solmate/src/auth/Owned.sol:39-43`

`ProtocolFees` uses immediate role-based controls with no in-contract delay mechanism:

- `owner` can instantly set `protocolFeeController` via `setProtocolFeeController`.
- `protocolFeeController` can immediately call:
  - `setProtocolFee(...)` (subject to max fee bounds), and
  - `collectProtocolFees(...)` to transfer accrued protocol fees to any recipient.

Relevant code:

```
function setProtocolFeeController(address controller) external onlyOwner {
 protocolFeeController = controller;
}

function setProtocolFee(...) external {
 if (msg.sender != protocolFeeController) revert InvalidCaller();
 ...
}

function collectProtocolFees(...) external returns (uint256 amountCollected) {
 if (msg.sender != protocolFeeController) revert InvalidCaller();
 amountCollected = (amount == 0) ? protocolFeesAccrued[currency] : amount;
 protocolFeesAccrued[currency] -= amountCollected;
 currency.transfer(recipient, amountCollected);
}
```

Concrete trace: after deployment with owner `0`, if `0` sets controller to `A`, `A` can in the next transaction update pool protocol-fee parameters (within validity bounds) and withdraw all currently accrued protocol fees by passing `amount=0`.

This is a governance/centralization trust assumption rather than a permissionless exploit. Also, “single-key” is deployment-dependent (owner/controller could be multisig contracts), but there is no built-in timelock/two-step acceptance in this codebase.

Impact:
compromised or malicious governance/controller authority can immediately change fee parameters and redirect protocol-fee revenue.

Recommendation:
operate these roles behind multisig + timelock (or equivalent delayed governance), and consider two-step controller changes for safer role handover.

LOW

StateLibrary is tightly coupled to PoolManager/Pool storage layout and can silently misread state on layout mismatch

Locations:

```
src/libraries/StateLibrary.sol:10-28
src/libraries/StateLibrary.sol:154-186
src/libraries/StateLibrary.sol:321-345
src/PoolManager.sol:79-94
src/libraries/Pool.sol:80-88
```

`StateLibrary` hardcodes storage assumptions for both the `PoolManager._pools` mapping slot and `Pool.State` member offsets, then performs raw `extsload` reads using computed slots.

What is true in code:

- `POOLS_SLOT` is hardcoded to `6` (`StateLibrary.sol:11`).
- Internal offsets are hardcoded (`FEE_GROWTH_GLOBAL0_OFFSET=1`, `LIQUIDITY_OFFSET=3`, `TICKS_OFFSET=4`, `TICK_BITMAP_OFFSET=5`, `POSITIONS_OFFSET=6`).
- Slot computation is manual (`keccak256(poolId, POOLS_SLOT)` and mapping-offset arithmetic).
- `Extsload.extsload` is a raw `sload` wrapper and does not validate schema/contract type (`src/Extsload.sol:10-14`).

Validation:

For the current `PoolManager` implementation, these constants are correct (`_pools` is indeed at slot 6 with current inheritance/storage ordering), so getters work today.

However, if storage layout differs (e.g., a manager implementation with one extra state variable before `_pools`, or changed `Pool.State` field order), `StateLibrary` will read wrong slots and decode incorrect values **without revert**. Example: if `_pools` base slot becomes 7, library still reads `keccak256(poolId,6)` while true pool state is at `keccak256(poolId,7)`.

Impact:

This is not an in-protocol exploit in current v4-core (core logic does not depend on `StateLibrary` for critical state transitions). The risk is primarily to downstream consumers/integrators that treat this as a generic `IPoolManager` state reader: silent misreads can propagate into incorrect accounting/decision logic.

Recommendation:

Treat `StateLibrary` as version-specific to canonical v4 `PoolManager` layout, and enforce this explicitly (docs + version pinning / manager codehash checks / explicit compatibility guardrails) so incompatible managers cannot be queried accidentally.

LOW

StateLibrary.getFeeGrowthInside accepts invalid tick ranges and can return wrapped fee-growth values

Location:

`src/libraries/StateLibrary.sol:295-319`

`StateLibrary.getFeeGrowthInside()` does not validate `tickLower < tickUpper` or `TickMath` bounds before performing arithmetic in an `unchecked` block.

Because `getTickFeeGrowthOutside()` reads arbitrary tick mapping slots, invalid inputs can produce meaningless subtraction order and wraparound outputs.

Example from repository-backed pool state (see `test/libraries/StateLibrary.t.sol` setup): after creating liquidity around `[-60,60]` and swapping, the tests show:

- `tickCurrent = -139`
- `feeGrowthOutside(-60) = 307621477895193619215525337320063 6`
- `feeGrowthOutside(60) = 0`

Calling `getFeeGrowthInside(manager, poolId, 60, -60)` takes branch `tickCurrent < tickLower` and computes:

- `0 - 3076214...` in `unchecked`, returning `2^256 - 3076214...` (near-`uint256.max`) instead of meaningful fee growth.

This is **not exploitable through v4-core's own liquidity accounting path** (core uses `Pool.getFeeGrowthInside` behind `Pool.checkTicks`), but it is a footgun for downstream on-chain/off-chain consumers that treat this helper as validated and feed user-controlled ticks.

Recommendation:

add explicit validation (or provide a checked variant) for ordered/in-range ticks before entering `unchecked` arithmetic, and document that invalid ranges return undefined modulo values if keeping current behavior.

LOW

`take`/`mint` and ERC6909 zero-recipient paths allow irreversible self-burn of value (while `settleFor(address(0))` does not persist)

Locations:

```
src/PoolManager.sol:290-295
src/PoolManager.sol:321-327
src/types/Currency.sol:40-53
src/ERC6909.sol:25-30
src/ERC6909.sol:35-43
src/ERC6909.sol:79-83
src/PoolManager.sol:112
src/PoolManager.sol:347-364
```

`PoolManager.take`, `PoolManager.mint`, and inherited ERC6909 transfer/mint logic accept `address(0)` recipients without validation.

What is real:

1. **`take(currency, address(0), amount)` can burn payouts**
   - In `take`, caller delta is debited first, then funds are transferred to `to`.
   - For native currency, `CurrencyLibrary.transfer` performs a raw `call` to `to`; with `to=address(0)`, ETH is sent to the zero address and is irrecoverable.
2. **`mint(address(0), id, amount)` can lock claim-backed value**
   - `mint` debits caller delta and mints ERC6909 claims to `to`.
   - Minting to `address(0)` creates claim balance at zero address; those claims cannot be moved/burned in practice because zero address cannot set operator/allowance or initiate transfers.
3. **ERC6909 direct transfers to zero also self-burn claims**
   - `transfer`/`transferFrom` credit `balanceOf[receiver][id]` with no zero check.
   - Sending claims to `address(0)` similarly strands claim rights.

Important correction:

The original finding's `settleFor(address(0))` lock scenario is **not persistent** when `paid > 0`:

- `_settle` credits delta to recipient (here zero), but unlock enforces `NonzeroDeltaCount == 0` at end.
- A positive delta on `address(0)` cannot be netted out by subsequent calls, so `unlock` reverts with `CurrencyNotSettled`, reverting the whole transaction atomically.
- Therefore `settleFor(address(0))` does not create a lasting stuck credit path.

Impact:

This is a **self-inflicted footgun** (user/integrator-controlled recipient), not a permissionless theft vector against other users. Still, it enables irreversible loss/lock of the caller's own value if zero address is passed accidentally or via unsafe integration defaults.

Recommendation:

Add explicit `address(0)` checks for recipient-like parameters where burn is not intended (`take.to`, `mint.to`) and/or provide explicit burn semantics/documentation. For ERC6909 claims, consider disallowing zero recipient transfers if claim burning is not intended via transfer semantics.

LOW

`collectProtocolFees` is callable while unlocked despite interface claiming it reverts when unlocked

Locations:

`src/interfaces/IProtocolFees.sol:39-41`

`src/ProtocolFees.sol:44-52`

`src/ProtocolFees.sol:59-60`

`src/PoolManager.sol:104-114`

`src/PoolManager.sol:390-393`

`IProtocolFees.collectProtocolFees` is documented with `@dev This will revert if the contract is unlocked`, but the implementation does not enforce that condition.

In `ProtocolFees.collectProtocolFees` the only gates are: 1) `msg.sender == protocolFeeController`, and 2) synced-currency protection for **non-native** currency (`!currency.isAddressZero() && getSyncedCurrency() == currency`).

There is no call to `_isUnlocked()` (even though `_isUnlocked` is declared in `ProtocolFees` and implemented in `PoolManager`).

As a result, if `protocolFeeController` is a contract invoked during `unlock` execution (e.g., unlock callback router or hook path), it can call `collectProtocolFees` mid-unlock and transfer protocol-fee assets during the unlocked phase. Native currency also bypasses the synced-currency guard by design of the condition.

Impact: this is primarily a spec/behavior mismatch and integration footgun. It can violate assumptions that fee collection is only possible when locked, and can introduce unexpected mid-unlock transfer/revert behavior in privileged callback flows. This is not a permissionless theft vector because triggering requires the privileged `protocolFeeController` role.

Recommendation:

- Either enforce the documented invariant by reverting when `_isUnlocked()` is true, or
- Update docs/comments to match actual behavior (collection allowed while unlocked except synced ERC20), including explicit native-currency semantics.

LOW

`settle` credits any intra-window ERC20 balance increase, allowing capture of rebasing/yield windfalls

Locations:

`src/PoolManager.sol:278-286`

`src/PoolManager.sol:347-364`

`src/libraries/CurrencyReserves.sol:27-31`

`src/libraries/CurrencyReserves.sol:21-24`

For ERC20 settlement, `PoolManager` snapshots manager balance in `sync(currency)` and later computes payment in `_settle` as:

- `reservesBefore = getSyncedReserves()`
- `reservesNow = currency.balanceOfSelf()`
- `paid = reservesNow - reservesBefore`

Then it credits `recipient` with `paid` via `_accountDelta`.

There is no authentication that the increase came from the settling payer (no `transferFrom` evidence, no payer binding). Therefore, any **balance increase that happens after `sync` and before `settle`** is treated as paid value.

Concrete trace (single `unlock` callback):

1. Manager holds 1,000 R
2. Attacker calls `sync(R)` ' reserves snapshot = 1,000
3. Token-side action increases manager balance to 1,100 without payer transfer (e.g., permissionless rebase/harvest-style update)
4. `settle()` credits attacker callback with `paid = 100`
5. `take(R, attackerEOA, 100)` withdraws 100 and brings callback delta back to zero
6. `unlock` completes (`NonzeroDeltaCount == 0`)

So the behavior is real.

Impact calibration: this generally diverts **unaccounted surplus/windfall** (e.g., rebasing/yield increase) rather than stealing already-accounted pool principal, because pool/protocol accounting did not include that increase. Still, it enables first-caller capture of value that many integrators/LPs may expect to remain with pool-held balances.

Recommended mitigations:

- Explicitly disallow or strongly warn against rebasing / balance-mutating tokens at integration level, or
- Use wrappers for rebasing assets, or
- Redesign settlement flow to bind credited amount to authenticated payer transfer semantics (architectural change).

LOW

Fee-growth floor rounding can leave small unallocated dust in PoolManager

Locations:

`src/libraries/Pool.sol:397-404`

`src/libraries/Pool.sol:463-476`

`src/libraries/Position.sol:91-97`

`src/PoolManager.sol:104-113`

`src/ProtocolFees.sol:54-56`

`Pool.swap()` and `Pool.donate()` credit LP fee growth with floor division (`simpleMulDiv`), while LP realization in `Position.update()` also uses floor division.

```
// Pool.sol
step.feeGrowthGlobalX128 += UnsafeMath.simpleMulDiv(step.feeAmount, Q128, liquidity);
state.feeGrowthGlobal0X128 += UnsafeMath.simpleMulDiv(amount0, Q128, liquidity);

// Position.sol
feesOwed0 = FullMath.mulDiv(deltaFeeGrowth, liquidity, Q128);
```

Because both stages round down, some paid fee/donation amount can remain unallocated in accounting. Concrete trace:

- Let in-range liquidity `L=3`, donate `amount0=1`.
- `feeGrowth = floor(Q128/3)`.
- Realized fees on poke: `floor(feeGrowth * 3 / Q128) = 0`.
- Donor paid 1 unit, LP realizes 0 for that update.

Across repeated accruals, this creates residual balance in PoolManager not attributed to LP fees or `protocolFeesAccrued`. The amount is cadence-dependent:

- If LP realizes once after many accruals, losses partially net out (e.g., 100 donations -> 99 realized, 1 residual in this example).
- If LP realizes after each accrual, the loss can crystallize more often.

So the original “1 unit per accrual action” claim is not universally true for deferred realization, but the underlying dust-stranding behavior is real.

LOW

Hook callback permissions are address-bit based; proxy/upgradeable hook deployments can silently misconfigure callback set if validation is omitted

Locations:

`src/interfaces/IHooks.sol:9-15`

`src/libraries/Hooks.sol:78-82`

`src/libraries/Hooks.sol:108-126`

`src/libraries/Hooks.sol:247-255`

`src/libraries/Hooks.sol:283-303`

`src/libraries/Hooks.sol:336-338`

`src/PoolManager.sol:117-127`

`v4-core` determines which hook callbacks to invoke exclusively from low bits of the hook address (`hasPermission`: `uint160(address(self)) & flag != 0`).

`PoolManager.initialize` only enforces `isValidHookAddress`, which checks structural validity (e.g., return-delta flag consistency and at least one flag/dynamic-fee) but does **not** enforce a developer-intended permission profile.

`validateHookPermissions` exists for this purpose, but is only an optional library check (`internal pure`) with constructor-oriented guidance; there is no protocol-level requirement that it must run.

As a result, in proxy/upgradeable hook setups, if deployers omit an initializer/runtime permission assertion and the proxy address bits do not match intended callbacks, operations proceed with callbacks silently skipped:

- `beforeSwap`/`afterSwap` call hooks only when corresponding bits are set.
- `before/afterModifyLiquidity` similarly gate on bits.
- When skipped, hook deltas stay zero and PoolManager does not account hook deltas.

So misconfigured pools can run without expected hook-side guards/accounting logic, with no explicit revert from the protocol.

Impact framing: This is primarily an integration/deployment footgun (self-misconfiguration), not a permissionless exploit against correctly configured pools.

Recommendation:

1. For proxy hooks, perform `validateHookPermissions` (or equivalent bit assertions) in an initializer and make it non-skippable.
2. Add explicit docs/examples for upgradeable deployments emphasizing that callback permissions are bound to hook address bits and are immutable for a fixed proxy address.
3. Optionally provide a reusable runtime assertion helper for post-deploy verification workflows.

LOW

Per-step protocol-fee flooring causes cumulative under-collection versus aggregated calculation

Locations:

`src/libraries/Pool.sol:381-394`

`src/libraries/Pool.sol:397-403`

`src/libraries/ProtocolFeeLibrary.sol:14-16`

`src/PoolManager.sol:233-238`

`src/ProtocolFees.sol:66-69`

`Pool.swap` computes protocol fee inside each swap-loop step, not on an aggregated swap total. For each step it does:

```
uint256 delta = (swapFee == protocolFee)
 ? step.feeAmount
 : (step.amountIn + step.feeAmount) * protocolFee / 1_000_000; // floor
step.feeAmount -= delta;
amountToProtocol += delta;
```

Because integer division floors each step, protocol accrual is: `sum_i floor(grossStepIn_i * protocolFee / 1e6)` instead of `floor(sum_i grossStepIn_i * protocolFee / 1e6)`.

So fragmented many-step swaps systematically under-collect protocol fees versus a one-shot computation. The truncated remainder stays in `step.feeAmount` and is credited to LP fee growth (`feeGrowthGlobalX128` update), while only `amountToProtocol` is sent to protocol accounting.

Concrete example at max protocol fee (1000 pips = 0.1%): if a swap is forced across 500 tiny steps with gross input 999 units/step, per-step collection is 0 each step (total 0), while one-shot on the same total input (499,500) is 499 units. The 499-unit gap is shifted away from protocol accrual.

This is an economic revenue-leakage issue (not direct theft of pool funds). It is most noticeable for low-decimal tokens and heavily fragmented tick layouts; for high-decimal assets the per-step loss is often negligible in token terms.

LOW

`donate(0,0)` is allowed, enabling zero-value donate hook execution and zero-amount Donate log spam

Locations:

`src/PoolManager.sol:255-275`

`src/libraries/Pool.sol:463-475`

`src/libraries/Hooks.sol:317-333`

`PoolManager.donate` does not enforce `amount0 > 0 || amount1 > 0`, and `Pool.donate` only checks `liquidity != 0`.

As a result, a caller can execute `donate(key, 0, 0, hookData)` successfully:

- `beforeDonate`/`afterDonate` hooks still run when donate hook flags are enabled (`Hooks.beforeDonate/afterDonate` are permission-gated only, not amount-gated).
- `Pool.donate(0,0)` returns `ZERO_DELTA` and does not update `feeGrowthGlobal0X128/feeGrowthGlobal1X128`.
- `_accountPoolBalanceDelta` receives zero deltas and performs no accounting changes.
- `Donate(poolId, sender, 0, 0)` is still emitted.

So zero-value calls can trigger donate-related hook side effects without transferring donated value, and can produce unlimited zero-amount Donate logs.

Impact in v4-core is behavioral/integration risk (not direct core fund loss): hooks or indexers that assume every donate is economically meaningful may be abused unless they validate amounts.

Recommendation: require `amount0 > 0 || amount1 > 0` in `PoolManager.donate` (or skip hook callbacks + event emission when both are zero), and document this expectation for hook implementers if zero-donations are intentionally supported.

LOW

`tickSpacingToMaxLiquidityPerTick` uses one extra tick bucket for most spacings, reducing per-tick liquidity capacity

Locations:

`src/libraries/Pool.sol:562-577`

`src/libraries/Pool.sol:162-169`

`src/libraries/TickMath.sol:45-49`

`Pool.tickSpacingToMaxLiquidityPerTick()` computes the minimum tick quotient with floor semantics for negative values:

```
let minTick := sub(sdiv(MIN_TICK, tickSpacing), slt(smod(MIN_TICK, tickSpacing), 0))
let maxTick := sdiv(MAX_TICK, tickSpacing)
let numTicks := add(sub(maxTick, minTick), 1)
```

For `tickSpacing` where `MIN_TICK % tickSpacing != 0`, this makes `minTick` one unit smaller than the usable-boundary quotient implied by `TickMath.minUsableTick()`/`maxUsableTick()` (which use truncation-toward-zero division before multiplying back by spacing). As a result, `numTicks` is `+1`, and `maxLiquidityPerTick` is slightly smaller than a usable-tick-count-based value.

Concrete trace (`tickSpacing = 32767`):

- Usable ticks: `-884709 .. 884709` (55 ticks)
- Current function count: quotients `-28 .. 27` (56 ticks)
- Current cap: `floor((2^128-1)/56) = 6076470837873901133274546561281575204`
- Usable-count cap: `floor((2^128-1)/55) = 61869521258352447902431746805776038 44`

In `modifyLiquidity`, adds revert when endpoint gross liquidity exceeds the computed cap:

```
if (state.liquidityGrossAfterLower > maxLiquidityPerTick) revert TickLiquidityOverflow(...);
if (state.liquidityGrossAfterUpper > maxLiquidityPerTick) revert TickLiquidityOverflow(...);
```

So liquidity additions between these two values are rejected earlier than necessary.

Impact:
conservative capacity reduction (no theft/accounting break). It causes caller-local `TickLiquidityOverflow` reverts for large additions and reduces maximum deployable liquidity per tick. Magnitude is tiny for common spacings (e.g. ~0.00056% at 10, ~0.0034% at 60, ~0.011% at 200) and up to ~1.79% at max allowed spacing 32767.

Recommendation:
derive `numTicks` from usable boundaries (equivalent to truncation-toward-zero quotient for `MIN_TICK`) so the cap matches actual usable ticks.

INFO

Unbounded unlock callback return data can gas-grief the caller/integration path

Locations:

`src/PoolManager.sol:104-110`

`src/test/ProxyPoolManager.sol:58-65`

`src/interfaces/callback/IUnlockCallback.sol:9`

`PoolManager.unlock` forwards control to `IUnlockCallback(msg.sender).unlockCallback(data)` and directly assigns the dynamic `bytes` return value to memory without any explicit max-length guard.

```
Lock.unlock();
result = IUnlockCallback(msg.sender).unlockCallback(data);
```

Because the return type is dynamic (`bytes memory`), very large returndata forces substantial memory expansion/copy work. For sufficiently large `N` (e.g., callback returning `new bytes(N)`), the call can run out of gas and revert.

This behavior is real, but impact is primarily **caller/integration scoped**:

- The callback target is always `msg.sender`, so a caller controls its own return size.
- Failed oversized attempts revert atomically; no persistent pool corruption or stuck global lock.
- Unrelated users/callers can still execute independent `unlock` transactions.

So this is best framed as an integration gas-footgun (unbounded returndata propagation), not a protocol-wide DoS.

Recommendation

If desired, harden `unlock` by capping accepted callback returndata length or ignoring callback return data (e.g., fixed empty return) to remove this gas-griefing surface for integrators.

INFO

`amountSpecified == int256.min` is accepted and can trigger `SafeCastOverflow` in exact-in edge paths

Locations:

`src/PoolManager.sol:192`

`src/PoolManager.sol:201-214`

`src/libraries/Hooks.sol:269-276`

`src/libraries/SwapMath.sol:63-66`

`src/libraries/SwapMath.sol:82`

`src/libraries/Pool.sol:376`

`src/libraries/SafeCast.sol:48-51`

`PoolManager.swap` validates only `amountSpecified != 0` before hooks, and does not reject `type(int256).min` either before or after `beforeSwap` modifies the value.

For `amountRemaining == type(int256).min`, `SwapMath.computeSwapStep` executes `uint256(-amountRemaining)` inside `unchecked`, which evaluates to `2^255` due two's-complement wraparound. In exact-in partial-fill logic, `feeAmount` can be derived from this `2^255` domain.

The concrete revert point is in `Pool.swap`:

```
amountSpecifiedRemaining += (step.amountIn + step.feeAmount).toInt256();
```

`SafeCast.toInt256` reverts for values `>= 2^255`.

A reachable case is exact-in swaps with `swapFee == 1e6` (100%) and non-zero liquidity: `amountRemainingLessFee` becomes 0, `feeAmount` becomes `2^255`, and the cast reverts with `SafeCastOverflow()`.

Hook path: a hook delta can set `amountToSwap` to `int256.min` (e.g., input `int256.min + 1`, delta `-1`). However, `hookDeltaSpecified` is only `int128`, so normal-sized user amounts cannot be pushed to `int256.min`.

Impact calibration:
this is revert-only behavior (atomic rollback, no partial state corruption, no fund theft). In practice it is mostly self-reverting input, or behavior in pools users chose to interact with (including hook-configured pools).

INFO

`resetCurrency()` leaves `RESERVES_OF_SLOT` stale; raw `exttload` consumers can misread transient reserves

Locations:

`src/libraries/CurrencyReserves.sol:21-24`

`src/PoolManager.sol:278-286`

`src/PoolManager.sol:347-361`

`src/Exttload.sol:10-13`

`src/libraries/TransientStateLibrary.sol:18-20`

`CurrencyReserves.resetCurrency()` clears only `CURRENCY_SLOT` and does not clear `RESERVES_OF_SLOT`.

```
function resetCurrency() internal {
 tstore(CURRENCY_SLOT, 0);
}
```

In `PoolManager`, both `sync(address(0))` and ERC20 `settle()` paths call `resetCurrency()`, so after these operations the transient state can be:

- `CURRENCY_SLOT == 0`
- `RESERVES_OF_SLOT == previous nonzero snapshot`

Because `Exttload.exttload` exposes raw transient slots publicly, same-transaction external readers can observe this stale reserves value if they query `RESERVES_OF_SLOT` directly.

Core protocol logic is protected: `_settle` only uses reserves when synced currency is nonzero, and `TransientStateLibrary.getSyncedReserves()` also guards on `CURRENCY_SLOT` before returning reserves.

So this is a real semantic/integration footgun for raw-slot consumers, not a direct PoolManager exploit.

Impact:

Integration correctness risk only. Third-party contracts/tools that read `RESERVES_OF_SLOT` directly (without checking `CURRENCY_SLOT`) may make incorrect same-tx decisions.

Recommendation:
Keep/document the invariant prominently (`RESERVES_OF_SLOT` meaningful only when `CURRENCY_SLOT != 0`), and consider a helper/API that enforces paired reads or clearing both slots when appropriate.

INFO

`bubbleUpAndRevertWith` leaves dynamic-bytes padding unwritten (latent memory-hygiene issue)

Location:

`src/libraries/CustomRevert.sol:91-117`

`CustomRevert.bubbleUpAndRevertWith` copies `returndatasize()` bytes into the `reason` field but sets the wrapped payload size using `encodedDataSize = ceil32(returndatasize())`. The gap between `returndatasize()` and `encodedDataSize` is included in the reverted byte length and is not explicitly zeroed.

```
let encodedDataSize := mul(div(add(returndatasize(), 31), 32), 32)
...
mstore(add(fmp, 0x84), returndatasize())
returndatacopy(add(fmp, 0xa4), 0, returndatasize())
// next field starts at add(fmp, add(0xa4, encodedDataSize))
```

So for non-32-byte-aligned revert reasons, `[fmp+0xa4+returndatasize(), fmp+0xa4+encodedDataSize)` is unwritten by this routine but still part of `revert(fmp, add(0xe4, encodedDataSize))`.

In current v4-core call paths this is generally benign because `fmp` is taken from a monotonic allocator frontier (no backward `mstore(0x40, ...)` in production paths), so those bytes are typically zero anyway. However, under non-standard memory reuse (e.g., unsafe assembly rewinding 0x40), stale bytes could appear in raw revert payload padding.

INFO

`extsload(bytes32,uint256)` lacks bounds checks: `nSlots<<5` can wrap, producing malformed returndata or OOG; array overloads remain linear-cost

Locations:

```
src/Extsload.sol:18-37
```

```
src/Extsload.sol:42-63
```

```
src/Exttload.sol:18-37
```

`Extsload.extsload(bytes32 startSlot, uint256 nSlots)` performs unchecked `length = shl(5, nSlots)` and uses an unconditional loop with `return(start, sub(end, start))`.

Confirmed concrete behaviors:

- `nSlots=0`: loop still executes once (one extra `sload`) but returns canonical 0x40-byte empty array encoding.
- `nSlots=2**251`: `length` wraps to 0, so returned bytes are only 64 bytes (`offset=0x20`, `length=2**251`, no payload). Low-level call can succeed, but Solidity decoding as `bytes32[]` reverts due length/data mismatch.
- `nSlots=2**251-3`: `end` wraps below `start`; `sub(end,start)` becomes near-`2**256`, so `RETURN` attempts enormous memory range and the call OOGs.

`extsload(bytes32[] calldata)` and `exttload(bytes32[] calldata)` also have unbounded linear loops, but their `slots.length` overflow branch is not practically reachable because ABI calldata must physically include `32*slots.length` bytes (bounded by calldata size/gas). So those paths are realistically a linear-cost resource issue, not shift-overflow exploitation.

Impact framing:

- No PoolManager state corruption/fund risk (view-only helpers).
- Main risk is integrator/RPC misuse: if an external contract forwards untrusted `nSlots` and decodes `bytes32[]`, attacker-controlled inputs can force revert/OOG in that caller path.

- v4-core internal helper usage is fixed-size (`StateLibrary` uses constants 2/3; `TransientStateLibrary` uses single-slot `exttload`), so core protocol flows are not directly exposed to this pattern.

Recommendation:

- Add explicit upper bounds for `nSlots` and `slots.length` (defensive for future integrations).
- For `extsload(bytes32,uint256)`, revert on `nSlots > type(uint256).max / 32` (or a much smaller practical cap) before computing `shl(5, nSlots)`.
- Consider an explicit maximum response size to avoid pathological memory/returndata behavior.

INFO

Before-hooks receive unvalidated numeric inputs (initialize ticks/price limits validated later in core)

Locations:

`src/PoolManager.sol:130-134`

`src/PoolManager.sol:155-159`

`src/PoolManager.sol:201-206`

`src/libraries/Pool.sol:97-104`

`src/libraries/Pool.sol:143-151`

`src/libraries/Pool.sol:317-335`

`src/libraries/Hooks.sol:251-279`

`PoolManager` invokes `before*` hooks before core numeric validation for several actions, so hook contracts can be called with values that are invalid per pool math constraints.

Verified paths:

1. **initialize**
   - `PoolManager.initialize` calls `key.hooks.beforeInitialize(key, sqrtPriceX96)` first, then `_pools[id].initialize(...)`.
   - Bounds validation for `sqrtPriceX96` occurs inside `Pool.initialize` via `TickMath.getTickAtSqrtPrice`.
   - Concrete case: `sqrtPriceX96 = TickMath.MAX_SQRT_PRICE` reaches hook first, then reverts with `InvalidSqrtPrice` in pool logic.
2. **modifyLiquidity**
   - `PoolManager.modifyLiquidity` calls `beforeModifyLiquidity` before `pool.modifyLiquidity`.
   - Tick validation (`checkTicks`) is inside `Pool.modifyLiquidity`.
   - Concrete case: `tickLower == tickUpper` (e.g., 10/10) is delivered to hook first, then reverts with `TicksMisordered`.
3. **swap**
   - `PoolManager.swap` calls `beforeSwap` before entering `Pool.swap`.
   - `sqrtPriceLimitX96` validity is checked in `Pool.swap`.

- Concrete case: `zeroForOne=true`, `amountSpecified=-1`, `sqrtPriceLimitX96=TickMath.MIN_SQRT_PRICE` reaches hook first, then reverts with `PriceLimitOutOfBounds`.

Impact:

This is primarily an integration/expectation issue: hooks must treat all params as untrusted and validate defensively if needed. There is no persistent state corruption/fund loss from these failing calls because reverting pool checks unwind the full transaction.

Note:

There is also an edge case where `beforeSwap` can adjust amount to `0` (via return delta), causing `Pool.swap` to early-return before price-limit checks. That further confirms hook-visible inputs are not guaranteed to be core-validated when observed by hooks.

Recommendation:

Document explicitly in hook developer guidance that `before*` hooks receive pre-validation inputs and must not assume Uniswap core bounds have been enforced yet. Optionally, pre-validate selected fields in `PoolManager` if stricter hook input guarantees are desired.

INFO

IHooks docs imply PoolManager-only entry, but v4-core does not enforce callback caller authentication

Locations:

`src/interfaces/IHooks.sol:14`

`src/libraries/Hooks.sol:130-154`

`IHooks` documents that callbacks "Should only be callable by the v4 PoolManager" (`src/interfaces/IHooks.sol:14`), but this is not enforced by the interface or a shared production base hook in `src/`.

In core execution, `PoolManager`/`Hooks` performs low-level external calls to the hook (`Hooks.callHook`) and validates success/returned selector only; there is no framework-level requirement that hook implementations gate callback entry by `msg.sender`.

As a result, hook callback functions remain publicly callable `external` methods unless each hook implementation adds its own guard (e.g., `require(msg.sender == address(poolManager))`).

Concrete behavior exists in repo test hooks:

- Some hooks add explicit `onlyPoolManager` modifiers (e.g., `FeeTakingHook`, `DeltaReturningHook`).
- Others do not (e.g., `MockHooks`, `DynamicFeesTestHook`), and direct third-party calls execute callback side effects.

Impact is primarily an integration/documentation pitfall: hook authors who rely on the wording and omit caller checks can expose privileged hook logic to arbitrary callers. This is not a direct v4-core accounting exploit by itself, so `info` is appropriate.

INFO

IHooks liquidity after-hook docs misdescribe `delta` as post-hook although implementation passes pre-hook value

Locations:

```
src/interfaces/IHooks.sol:50
src/interfaces/IHooks.sol:81
src/libraries/Hooks.sol:218-241
src/PoolManager.sol:169-177
```

`IHooks.afterAddLiquidity` and `IHooks.afterRemoveLiquidity` comments state that `delta` already includes `hook delta`.

That is not how execution works:

1. `PoolManager.modifyLiquidity` computes `callerDelta = principalDelta + feesAccrued` before after-hook execution.
2. `Hooks.afterModifyLiquidity` passes that pre-hook `delta` into `IHooks.afterAddLiquidity/afterRemoveLiquidity`.
3. Only after hook return does it apply the returned `hookDelta` via `callerDelta = callerDelta - hookDelta`.

So the interface prose and runtime semantics diverge.

Why this matters:

This is primarily an integration/documentation risk: hook authors may reason about `delta` incorrectly and implement wrong accounting logic in custom hooks. Core PoolManager accounting itself remains consistent (the per-call invariant `finalCallerDelta + hookDelta == principal + fees` holds).

Recommendation:

Update `IHooks` NatSpec for `afterAddLiquidity`/`afterRemoveLiquidity` to explicitly define:

- incoming `delta` is **pre-hook** (`principal + feesAccrued`), and
- returned `BalanceDelta` is the hook delta applied afterward by PoolManager.

Optionally add a brief comment in `Hooks.afterModifyLiquidity` mirroring this to avoid future confusion.

INFO

`IHooks.beforeSwap` documentation contradicts actual fee-override encoding/validation

Locations:

`src/interfaces/IHooks.sol:102`

`src/libraries/LPFeeLibrary.sol:19`

`src/libraries/LPFeeLibrary.sol:25`

`src/libraries/LPFeeLibrary.sol:75-77`

`src/libraries/Hooks.sol:260-263`

`src/libraries/Pool.sol:300-302`

The finding is valid as a **documentation/spec inconsistency**.

`IHooks.beforeSwap` currently says the returned fee override is only used when: 1) pool is dynamic, 2) override flag `0x400000` is set, and 3) the returned value is `<= 1_000_000`.

Those conditions are mutually inconsistent if interpreted on the raw returned `uint24`, because `0x400000` (4,194,304) is already greater than `MAX_LP_FEE` (1,000,000).

Actual runtime behavior is different:

- `Hooks.beforeSwap` only parses the fee field for dynamic-fee pools (`key.fee.isDynamicFee()`).
- `Pool.swap` checks `isOverride()` on the returned value.
- If override is set, it applies `removeOverrideFlagAndValidate()` (mask first, then validate masked fee).
- If override is not set, it ignores the returned fee and uses stored `slot0.lpFee()`.

Concrete trace:

- Returned `500` (unflagged): `isOverride == false` ' fallback to stored fee.
- Returned `0x400000 | 500` (4,194,804): `isOverride == true`; masked fee is `500`, which is valid and used.

So the implementation is internally coherent, but the interface text can mislead hook authors/integrators about how to encode/validate override values.

Recommendation:

Update `IHooks.beforeSwap` docs to explicitly state: the override flag is included in the returned value; the protocol masks out `OVERRIDE_FEE_FLAG` before validating against `MAX_LP_FEE`.

INFO

`clear()` docs overstate irrecoverability: it burns caller delta but does not lock underlying assets permanently

Locations:

`src/interfaces/IPoolManager.sol:205-209`

`src/PoolManager.sol:309-318`

`src/PoolManager.sol:290-295`

`IPoolManager.clear` documentation says cleared currency is "locked in the contract permanently". The implementation does something narrower: it only zeroes the caller's positive transient delta.

```
int256 current = currency.getDelta(msg.sender);
int128 amountDelta = amount.toInt128();
if (amountDelta != current) revert MustClearExactPositiveDelta();
_accountDelta(currency, -(amountDelta), msg.sender);
```

No token/native transfer occurs in `clear()`. So the manager's actual token balance is unchanged by clear itself.

Concrete trace:

1. User A accrues `+5` delta (e.g., via `sync` + transfer + `settle`).
2. User A calls `clear(currency, 5)`: A's delta becomes `0`, but manager token balance still includes the 5 units.
3. Later, User B (in another unlock/tx) with a valid positive delta can call `take(currency, to, amount)`, which transfers from the same pooled manager balance:

```
_accountDelta(currency, -(amount.toInt128()), msg.sender);
currency.transfer(to, amount);
```

So cleared funds are non-retrievable for the clearer, but not necessarily globally non-retrievable at contract level; economically they behave as unassigned surplus/donation in pooled reserves.

Impact:

Documentation/semantic mismatch that can mislead integrators about asset fate. No direct unauthorized value extraction by itself.

Recommendation:

Reword `clear` docs to state that it clears the caller's claim only; underlying assets remain in PoolManager balances and may be used in later legitimate payouts/settlement flows.

INFO

`IPoolManager.unlock` NatSpec overstates locked-state callable surface

Locations:

`src/interfaces/IPoolManager.sol:108-113`

`src/PoolManager.sol:277-287`

`src/ProtocolFees.sol:29-57`

`src/ERC6909.sol:25-64`

`IPoolManager.unlock` documentation states that the only functions callable without unlocking are `initialize` and `updateDynamicLPFee`.

That statement is inaccurate for the actual `PoolManager` callable surface while locked:

- `sync(Currency)` is externally callable without `onlyWhenUnlocked`.
- Protocol fee functions (`setProtocolFeeController`, `setProtocolFee`, and also `collectProtocolFees`) have role checks but no lock-state check.
- ERC-6909 methods inherited by `PoolManager` (`transfer`, `transferFrom`, `approve`, `setOperator`) are callable without lock-state checks.

So the implementation allows multiple additional non-unlock entrypoints in locked state.

Impact is documentation/integration correctness: integrators/hook authors relying on this NatSpec can form incorrect assumptions about what can be called outside the `unlock` callback flow. No direct exploit or fund-loss path was identified from this mismatch alone.

Recommendation: update `unlock` NatSpec to avoid exhaustive enumeration (or enumerate accurately), e.g. clarify that *delta-accounting pool actions* require unlock, while other admin/token/view utility functions may remain callable when locked.

INFO

`unlockCallback` caller authentication is left to integrators, creating an integration footgun

Locations:

`src/PoolManager.sol:104-111`

`src/interfaces/callback/IUnlockCallback.sol:4-9`

`src/interfaces/IPoolManager.sol:108-113`

`PoolManager.unlock` always calls back into the caller contract via `IUnlockCallback(msg.sender).unlockCallback(data)`.

```
// PoolManager.unlock
Lock.unlock();
result = IUnlockCallback(msg.sender).unlockCallback(data);
```

The callback interface/docs only state that it is "Called by the pool manager" but do not explicitly require implementers to authenticate the caller inside `unlockCallback`.

As a result, downstream routers/periphery contracts that implement `unlockCallback` without a guard such as `require(msg.sender == address(poolManager))` can be directly invoked by arbitrary EOAs, executing whatever router logic is placed in that function.

Important calibration: in this repository, this is primarily an integration/documentation risk (no production non-test callback implementer is exposed), and PoolManager's lock still blocks direct execution of lock-gated core methods outside `unlock()`. So this is best classified as **Info**.

Recommendation:

- Update interface/docs to explicitly require caller authentication in all callback implementations.
- Provide/encourage a reusable base helper (e.g., `onlyPoolManager`) for callback contracts.
- Keep secure examples in reference periphery code.

INFO

`Initialize.fee` docs omit dynamic-fee sentinel (`0x800000`) semantics

Locations:

`src/interfaces/IPoolManager.sol:54`

`src/PoolManager.sol:128-140`

`src/types/PoolKey.sol:16-17`

`src/libraries/LPFeeLibrary.sol:14-55`

`PoolManager.initialize` emits the raw `key.fee` in `Initialize`, but the interface doc for the event describes `fee` only as a numeric swap fee (hundredths of a bip).

For dynamic-fee pools, `PoolKey.fee` can be the sentinel `0x800000` (not a literal fee value). In that path:

- `lpFee = key.fee.getInitialLPFee()` returns `0` for dynamic pools.
- pool state is initialized with `slot0.lpFee = 0`.
- event still emits `key.fee` (`0x800000`).

So `Initialize.fee` can be a mode sentinel, not always a numeric fee.

This can mislead off-chain consumers that parse only `IPoolManager.Initialize` docs and treat `fee` as always numeric, causing incorrect analytics/validation for dynamic pools. This is documentation/integration confusion only (no direct on-chain fund-loss path).

Recommendation

Update `IPoolManager.Initialize` event parameter docs to explicitly state that `fee` may be either: 1) a static fee value (`<= 1_000_000`), or 2) the dynamic-fee sentinel `0x800000` indicating dynamic LP fee mode. Also mention that actual active LP fee is sourced from pool state / swap-time logic, not necessarily the emitted sentinel value.

INFO

LiquidityMath.addDelta NatSpec incorrectly describes return value

Location:

`src/libraries/LiquidityMath.sol:9`

`LiquidityMath.addDelta(uint128 x, int128 y)` returns the **updated liquidity value** (`x + y`, with overflow/underflow revert), but its NatSpec says `@return z The liquidity delta`.

Implementation evidence:

- `z := add(...x..., signextend(15, y))` then revert if upper 128 bits are nonzero (`shr(128, z)`), which enforces `z` is the valid post-update `uint128` liquidity.
- Concrete traces: `x=100,y=10 => z=110`; `x=100,y=-10 => z=90`; `x=0,y=-1` reverts; `x=type(uint128).max,y=1` reverts.

In-repo usage is correct (return assigned as new absolute liquidity in `Position.update` and `Pool` paths), so this is not an exploitable issue in deployed v4-core. Impact is documentation clarity / downstream footgun risk for copied integrations that might treat returned `z` as a delta and double-apply updates.

INFO

`donate()` rewards instantaneous in-range liquidity, enabling JIT/MEV capture of donation value

Locations:

`src/PoolManager.sol:255-275`

`src/libraries/Pool.sol:463-475`

`src/interfaces/IPoolManager.sol:164-166`

`PoolManager.donate()` forwards to `Pool.donate()`, which distributes donation amounts purely via the current `state.liquidity` snapshot:

```
uint128 liquidity = state.liquidity;
if (liquidity == 0) revert NoLiquidityToReceiveFees();
state.feeGrowthGlobal0X128 += amount0 * Q128 / liquidity;
state.feeGrowthGlobal1X128 += amount1 * Q128 / liquidity;
```

Because liquidity can be modified while unlocked, an actor can execute (or sandwich) the sequence `modifyLiquidity(+L)` -> `donate()` -> `modifyLiquidity(-L)` and earn donation share proportional to temporary in-range liquidity.

Concrete trace: if existing in-range liquidity is 1,000 and attacker adds 9,000 before a 100-token donation, donation growth is computed over 10,000 liquidity and attacker later realizes about `100 * 9000 / 10000 = 90` tokens when updating/removing. The attacker does not receive prior historical growth because `Position.update` checkpoints fee growth on add.

Impact is incentive dilution/opportunity loss for incumbent LPs (not pool insolvency or direct reserve theft). This is a known and documented behavior in `IPoolManager` (`donate` can be frontrun by JIT liquidity), so severity is informational.

INFO

PoolManager has no generic sweep path for unaccounted ERC20/native balances (mis-sent funds may be unrecoverable)

Locations:

`src/PoolManager.sol:278-287`

`src/PoolManager.sol:290-306`

`src/PoolManager.sol:347-364`

`src/ProtocolFees.sol:43-57`

`src/interfaces/IPoolManager.sol:188-226`

`PoolManager` does not implement a generic rescue/sweep mechanism for arbitrary balances held by the contract, and the existing outflow paths are accounting-gated.

What is true in code:

- `take()` is gated by caller delta accounting (`_accountDelta(currency, -amount, msg.sender)`) and only usable within unlock flow; caller must finish with net-zero deltas or `unlock` reverts (`CurrencyNotSettled`) (`PoolManager.sol:104-113`, `290-295`).
- `settle()` for ERC20 only credits the delta `balanceNow - reservesBefore` after `sync()`; pre-existing balance is treated as baseline, not credited (`PoolManager.sol:278-287`, `347-364`).
- `settle()` for native credits only `msg.value` (not pre-existing contract ETH balance) when synced currency is zero (`PoolManager.sol:351-354`).
- `collectProtocolFees()` can only transfer up to tracked `protocolFeesAccrued[currency]`; attempting to collect untracked balance reverts via underflow (`ProtocolFees.sol:54-56`).
- `IPoolManager` exposes no admin/user generic `sweep/rescue/recover` method (`IPoolManager.sol:188-226`).

Concrete scenario:

If 100 USDC is transferred directly to `PoolManager` and 1 ETH is force-sent, those balances are not automatically mapped to user delta or protocol-fee accounting. A caller cannot simply withdraw them unless they already have independent positive accounting rights (or repay in-tx to net out deltas).

Impact:

Primarily operational/UX: mistaken direct transfers (or forced ETH) can become unattributed and not directly recoverable by the sender through protocol interfaces. This is not a direct theft vector.

Recommendation:

If desired, add a tightly controlled rescue path (e.g., governance-only with safeguards and clear policy), or explicitly document that direct transfers/forced ETH are unsupported and may be unrecoverable by sender.

INFO

`NonzeroDeltaCount.decrement()` wraps on zero and can falsely trigger `CurrencyNotSettled` if counter/delta invariant is ever broken

Locations:

`src/libraries/NonzeroDeltaCount.sol:28-33`

`src/PoolManager.sol:367-376`

`src/PoolManager.sol:104-113`

`NonzeroDeltaCount.decrement()` performs unchecked assembly subtraction:

```
let count := tload(NONZERO_DELTA_COUNT_SLOT)
count := sub(count, 1)
tstore(NONZERO_DELTA_COUNT_SLOT, count)
```

So `count == 0` wraps to `2^256 - 1` instead of reverting. In `PoolManager._accountDelta`, the `next == 0` branch always calls `decrement()` with no guard on the counter value:

```
if (next == 0) {
 NonzeroDeltaCount.decrement();
} else if (previous == 0) {
 NonzeroDeltaCount.increment();
}
```

If the transient counter ever drifts from actual nonzero deltas (e.g., forced inconsistent transient state), a call that transitions a delta `1 -> 0` will underflow the counter. Then `unlock()` end check reverts:

```
if (NonzeroDeltaCount.read() != 0) CurrencyNotSettled.selector.revertWith();
```

This can cause an `unlock` attempt to fail even when per-currency deltas are actually netted to zero.

Impact calibration: in the current codebase, this requires prior invariant corruption; there is no permissionless path to write delta slots or `NONZERO_DELTA_COUNT_SLOT` outside `_accountDelta`. Therefore this is best treated as a defensive fragility / invariant-hardening issue (Info), not an exploitable bug.

Recommendation: add a defensive assertion/guard (e.g., check `count > 0` before decrement, or assert `(previous != 0)`-aligned counter invariants in debug/production-safe form) to fail fast with clearer diagnostics and prevent wraparound behavior under future integration changes.

INFO

ParseBytes helpers are not abi.decode-equivalent on short inputs (caller must enforce minimum length)

Locations:

`src/libraries/ParseBytes.sol:9-28`

`src/libraries/Hooks.sol:150-167`

`src/libraries/Hooks.sol:257-267`

`ParseBytes.parseSelector`, `parseReturnDelta`, and `parseFee` use fixed-offset `mload` reads with no internal `result.length` validation. Their inline comments state they are `abi.decode`-equivalent, but `abi.decode` would revert on undersized input while these helpers return memory contents.

Example (concrete): with `bytes result = hex""` (length 0), `parseSelector()` executes `mload(add(result, 0x20))` and returns `0x00000000` (or whatever is in memory), instead of reverting.

Current core usage is protected: `Hooks.callHook` enforces `result.length >= 32` before `parseSelector`; `callHookWithReturnDelta` requires `result.length == 64` before `parseReturnDelta`; `beforeSwap` requires `result.length == 96` before `parseFee`/`parseReturnDelta`. So this is not currently exploitable in v4-core hook flows.

Impact is therefore limited to API sharp-edge / maintenance risk: future reuse or refactoring without explicit length checks could silently mis-parse hook responses.

Recommendation: either (a) add explicit length checks in `ParseBytes` to match `abi.decode` semantics, or (b) update comments/docs to clearly state required minimum lengths and caller responsibility.

INFO

Pool/Slot0 fee storage mutators do not enforce fee bounds and rely on caller-side validation

Locations:

`src/libraries/Pool.sol:97-115`

`src/types/Slot0.sol:78-93`

`src/libraries/SwapMath.sol:63-66`

`src/libraries/Pool.sol:307-313`

`Pool.initialize`, `Pool.setLPFee`, and `Pool.setProtocolFee` persist fee values directly into `slot0` without local range checks, and `Slot0Library.setLpFee` / `setProtocolFee` only bit-pack values.

So the invariant (`lpFee <= 1_000_000`, protocol components `<= 1000`) is currently enforced only by upstream call sites:

- `PoolManager.initialize` -> `key.fee.getInitialLPFee()`
- `PoolManager.updateDynamicLPFee` -> `newDynamicLPFee.validate()`
- `ProtocolFees.setProtocolFee` -> `isValidProtocolFee()`

This means the current deployed paths are safe, but the pool library itself is not self-defensive against invalid fee writes.

Why this matters if the invariant is ever bypassed:

- `SwapMath.computeSwapStep` exact-input path assumes `feePips <= MAX_SWAP_FEE` and computes `MAX_SWAP_FEE - feePips` inside an `unchecked` block.
- With an invalid stored fee (e.g., `feePips = 1_000_001`), this subtraction wraps and fee math becomes nonsensical (e.g., near-zero effective fee or abnormal exact-input accounting behavior), while exact-output is only guarded by `InvalidFeeForExactOut`.

Impact framing: this is a defensive-invariant hardening issue, not a current permissionless exploit in v4-core as shipped.

Recommendation:

- Add validation at storage mutation boundaries (`Pool.initialize`, `Pool.setLPFee`, `Pool.setProtocolFee`) or in `Slot0` setters, so invariants hold regardless of future refactors/integrations.

INFO

Permissionless first initializer can front-run and set the pool's initial price for a given PoolKey

Locations:

`src/PoolManager.sol:117-135`

`src/libraries/Pool.sol:97-104`

`src/types/PoolId.sol:10-15`

`PoolManager.initialize` is externally callable without role checks, and pool identity is `PoolId = keccak256(abi.encode(PoolKey))` (price is not part of the id). Therefore, for any not-yet-initialized `PoolKey`, whichever transaction lands first sets `slot0.sqrtPriceX96`.

`Pool.initialize` then permanently blocks re-initialization for that same pool id:

- first call: writes `slot0` with chosen `sqrtPriceX96`
- later calls: revert with `PoolAlreadyInitialized` when `slot0.sqrtPriceX96() != 0`

So an observer can front-run a pending initialization transaction and lock in a different valid starting price for that exact key. This is an initialization-griefing / pool-squatting behavior, not a direct theft vector.

Impact is primarily operational/integration risk (e.g., deployers or integrators assuming they control initial price). It does not directly compromise protocol funds.

Recommendation: keep this behavior clearly documented as a permissionless-design property; integrators should not assume initialization exclusivity and should verify on-chain pool state before subsequent actions. Where controlled initialization is needed, use pool-specific hook logic (e.g., `beforeInitialize` authorization) or private orderflow/deployment procedures.

INFO

Pool/TickBitmap library code assumes tickSpacing > 0; zero value would silently collapse tick indexing and desynchronize bitmap

Locations:

`src/libraries/Pool.sol:562-578`

`src/libraries/Pool.sol:143-178`

`src/libraries/TickBitmap.sol:47-74`

`Pool`/`TickBitmap` contain assembly paths that do not defensively reject `tickSpacing == 0`.

What is true in code:

- `Pool.tickSpacingToMaxLiquidityPerTick` uses `sdiv/smod` in assembly and comments that spacing is never zero.
- With `tickSpacing=0`, EVM semantics are non-reverting (`sdiv(x,0)=0`, `smod(x,0)=0`), so:
  - `minTick=0`, `maxTick=0`, `numTicks=1`, and result becomes `type(uint128).max`.
- `TickBitmap.flipTick` also uses `smod/sdiv`:
  - misalignment check is bypassed (`smod(tick,0)==0`),
  - normalized tick becomes `0`,
  - every flip toggles the same bitmap bit (`wordPos=0`, `bitPos=0`).

In `Pool.modifyLiquidity`, this causes tick bitmap state to desynchronize from actual initialized ticks when zero spacing is used (single operation with `tickLower=-120`, `tickUpper=120` flips both ticks but toggles the same bit twice, ending with bitmap unset while both ticks are initialized).

Realism / reachability in this repo:

Canonical `PoolManager` flow prevents this today:

- `initialize` enforces `tickSpacing >= TickMath.MIN_TICK_SPACING` (MIN=1), rejecting zero.
- `PoolId` includes `tickSpacing`, so passing `tickSpacing=0` to `modifyLiquidity/swap` points to a different (uninitialized) pool and reverts.

So this is **not currently permissionless-exploitable** in deployed `PoolManager` flow, but it is a latent defensive-safety footgun if future integrations/refactors bypass the invariant.

Recommendation:

Add explicit defensive checks at library consumption points (or inside helper functions) for `tickSpacing > 0` before assembly math/bitmap operations, so invariant violations fail fast rather than silently corrupting state.

INFO

`Pool.setLPFee` NatSpec implies dynamic-fee-only enforcement, but restriction is caller-enforced

Locations:

`src/libraries/Pool.sol:111-115`

`src/PoolManager.sol:338-345`

`src/libraries/LPFeeLibrary.sol:30-32`

`Pool.setLPFee` is documented as “Only dynamic fee pools may update the lp fee”, but the function body only checks pool initialization and writes `slot0.lpFee`.

```
function setLPFee(State storage self, uint24 lpFee) internal {
 self.checkPoolInitialized();
 self.slot0 = self.slot0.setLpFee(lpFee);
}
```

The dynamic-fee invariant is actually enforced in `PoolManager.updateDynamicLPFee` via:

- `key.fee.isDynamicFee()`
- `msg.sender == address(key.hooks)`
- `newDynamicLPFee.validate()`

So current production behavior is safe, but the NatSpec on `Pool.setLPFee` overstates what that function itself enforces. This creates a maintenance/integration footgun: if a new internal call path is added (or a derived manager exposes a helper) and relies on `Pool.setLPFee` NatSpec instead of duplicating guards, static-fee pools could have LP fee mutated.

Impact is documentation/semantic mismatch and future bug risk, not an immediate exploitable issue in current code.

INFO

PoolId hashing hardcodes current PoolKey word length (0xa0), creating a migration/upgrade compatibility invariant

Locations:

`src/types/PoolId.sol:11-15`

`src/types/PoolKey.sol:11-21`

`src/PoolManager.sol:93`

`src/PoolManager.sol:132-135`

`PoolIdLibrary.toId()` hashes a fixed 160-byte slice of `PoolKey` memory:

```
assembly ("memory-safe") {
 poolId := keccak256(poolKey, 0xa0)
}
```

With the current 5-field static `PoolKey`, this matches `keccak256(abi.encode(poolKey))` (also covered by `test/libraries/PoolId.t.sol`). However, this creates a maintenance invariant: pool IDs depend on this exact struct shape/encoding assumption.

If a future implementation changes ID derivation semantics (e.g., altered struct fields/order/encoding assumptions) while expecting continuity of old pool storage, the same logical key could map to a different `PoolId`. Since pool state is keyed by `mapping(PoolId => Pool.State) _pools`, lookups would target a different branch, and pool operations would observe uninitialized state.

Impact:

- **Current codebase:** no direct exploit path; behavior is correct as deployed.
- **Future migration/upgrade risk:** operational compatibility hazard (possible pool-state lookup breakage if ID derivation changes).

Recommendation:

Treat `toId` as a strict backward-compatibility contract and document it as such. For any future migration/upgrade architecture:

1. Preserve exact ID derivation for legacy pools, or

2. Introduce explicit versioned/migration mapping logic with compatibility tests proving old keys resolve to expected IDs.

(Severity kept at **Info** because this is a forward-compatibility footgun, not a present permissionless exploit.)

INFO

Unchecked protocol fee accrual can wrap `protocolFeesAccrued` modulo 2^256

Locations:

`src/ProtocolFees.sol:66-69`

`src/PoolManager.sol:233-239`

`protocolFeesAccrued` is incremented in an `unchecked` block:

```
function _updateProtocolFees(Currency currency, uint256 amount) internal {
 unchecked {
 protocolFeesAccrued[currency] += amount;
 }
}
```

`PoolManager._swap` calls this when `amountToProtocol > 0`:

```
if (amountToProtocol > 0) _updateProtocolFees(inputCurrency, amountToProtocol);
```

So if `protocolFeesAccrued[currency]` is near `type(uint256).max`, a positive `amountToProtocol` will wrap silently (modulo `2^256`) instead of reverting.

Concrete trace: with stored value `type(uint256).max - 9` and `amountToProtocol = 10`, new stored value becomes `0`.

Impact:
This is an accounting correctness issue: after wrap, `collectProtocolFees` can only collect up to the wrapped value because it uses `protocolFeesAccrued` as the source of truth (`amountCollected` then checked subtraction). Any overflowed high bits are not recoverable via protocol-fee collection logic.

In practice, reaching this state requires extreme/unrealistic preconditions (very large uncollected fee accumulation, protocol fee enabled by controller, and prolonged non-collection), so practical exploitability is negligible.

Recommendation:

If strict non-wrapping accounting is desired, replace unchecked addition with checked arithmetic (or explicit bound checks and revert near capacity). If wrapping is intentional, document this invariant clearly and monitor/collect before approaching bounds.

INFO

SqrtPriceMath has non-uniform revert behavior across helper paths and asymmetric zero-price validation

Locations:

`src/libraries/SqrtPriceMath.sol:86-120`

`src/libraries/SqrtPriceMath.sol:53-72`

`src/libraries/SqrtPriceMath.sol:188-211`

`src/libraries/SqrtPriceMath.sol:234-254`

`src/libraries/SafeCast.sol:16-19`

`src/libraries/FullMath.sol:14-31`

`SqrtPriceMath` exposes mixed failure modes across similar helper APIs:

- `getNextSqrtPriceFromAmount1RoundingDown(..., add=true)` returns `(uint256(sqrtPX96) + quotient).toUint160()`, so oversized sums revert via `SafeCastOverflow` (from `SafeCast.toUint160`) rather than a `SqrtPriceMath` custom error.
- `getNextSqrtPriceFromAmount0RoundingUp(..., add=false)` can fail in three different ways depending on boundary condition: `PriceOverflow` (pre-check), generic `require` revert from `FullMath.mulDiv` (`denominator <= prod1`), or `SafeCastOverflow` on final downcast.
- `getAmount0Delta` explicitly rejects zero lower price (`InvalidPrice`), but `getAmount1Delta(uint160,uint160,uint128,bool)` does not reject zero endpoints and returns computed values.

Concrete traces:

- `getNextSqrtPriceFromAmount1RoundingDown(type(uint160).max-1, 1024, 1024, true)` => sum exceeds `uint160`, reverts `SafeCastOverflow`.
- `getAmount1Delta(0, Q96, 1e18, true)` => returns `1e18` (no revert), while `getAmount0Delta(0, Q96, 1e18, ...)` reverts `InvalidPrice`.

Important calibration note: for `getAmount1Delta` specifically, `FullMath.mulDiv`'s `require(denominator > prod1)` is not reachable under its type bounds (`uint128 * uint160 / 2^96`), so this is not a practical failure mode at that callsite.

In current protocol flows, pool initialization and swap/liquidity entry checks constrain prices and callers do not branch on specific revert selectors, so this is primarily a consistency/diagnostic issue rather than an exploit path.

INFO

StateLibrary getters return zero values for uninitialized pools and do not enforce initialization checks

Locations:

`src/libraries/StateLibrary.sol:40-63`

`src/libraries/StateLibrary.sol:180-188`

`src/Extsload.sol:10-14`

`StateLibrary` read helpers are raw storage readers and intentionally do not verify pool initialization.

- `getSlot0` computes the pool state slot and decodes `manager.extsload(stateSlot)` directly.
- `getLiquidity` computes the liquidity slot and returns `uint128(uint256(manager.extsload(slot)))`.
- `Extsload.extsload` is a direct `sload(slot)` wrapper.

For an uninitialized/nonexistent `poolId`, the mapping storage is unwritten, so `sload` returns `0x0`. This causes:

- `getSlot0` => `sqrtPriceX96=0`, `tick=0`, `protocolFee=0`, `lpFee=0`
- `getLiquidity` => `0`

Core protocol safety is not bypassed by this (mutating PoolManager paths call `checkPoolInitialized()`), but downstream consumers that accept user-supplied `poolId` and assume getters always return initialized data can mis-handle values (e.g., misquote at zero price or hit division-by-zero in follow-on math).

Recommendation: document this contract-level invariant explicitly in StateLibrary/NatSpec and require callers to validate initialization (commonly `sqrtPriceX96 != 0`) before using values for economic logic. Optional helper methods that revert on uninitialized pools would reduce integration footguns.

INFO

StateLibrary trusts caller-supplied `IPoolManager` and decodes unverified `extsload` responses

Locations:

`src/libraries/StateLibrary.sol:40-63`

`src/libraries/StateLibrary.sol:154-171`

`src/libraries/StateLibrary.sol:180-188`

`src/libraries/StateLibrary.sol:295-317`

`StateLibrary` getters make direct external calls to the provided `manager` and decode returned words without validating that `manager` is a real/canonical PoolManager.

Examples:

- `getSlot0` calls `manager.extsload(stateSlot)` and directly decodes packed fields (sqrtPrice, tick, protocolFee, lpFee).
- `getFeeGrowthGlobals` and `getLiquidity` similarly trust `extsload` return data.
- `getFeeGrowthInside` aggregates multiple such reads and performs `unchecked` arithmetic, so inconsistent spoofed reads can produce wrapped outputs.

If an integrating contract allows users to supply `manager`, this can cause: 1) revert/DoS when `manager` is an EOA or non-implementer (ABI decode failure from empty/invalid returndata), or 2) spoofed state when `manager` is a malicious contract implementing `extsload`.

In this v4-core repository, production contracts do not use `StateLibrary` in state-transition paths, so this does not directly compromise PoolManager accounting/funds. The risk is an integration sharp edge for downstream consumers that treat `manager` as untrusted input.

INFO

StateLibrary position tuple overload silently returns zero for malformed owner/tick inputs

Locations:

`src/libraries/StateLibrary.sol:227-239`

`src/libraries/StateLibrary.sol:251-265`

`src/libraries/Position.sol:48-60`

`StateLibrary.getPositionInfo(manager, poolId, owner, tickLower, tickUpper, salt)` does not validate tuple invariants before lookup. It always computes `positionKey = Position.calculatePositionKey(...)` and then performs a raw `extsload` of `pools[poolId].positions[positionKey]`.

Because this path is pure key derivation + storage read, malformed tuples (e.g., wrong owner, reversed ticks, out-of-range int24 values) do **not** revert here; they resolve to a different key and return zeroed `Position.State` if that slot is unset.

```
bytes32 positionKey = Position.calculatePositionKey(owner, tickLower, tickUpper, salt);
(liquidity, feeGrowthInside0LastX128, feeGrowthInside1LastX128) = getPositionInfo(manager,
poolId, positionKey);
```

Core write paths still enforce valid position creation (`owner = msg.sender`, `checkTicks` in `Pool.modifyLiquidity`), so this is not an in-protocol fund-risk bug. The risk is API-footgun behavior for downstream integrators that treat this helper as a validated existence/authorization check and pass untrusted tuple inputs.

Recommendation:

document explicitly that inputs are caller-validated and malformed tuples will return zeroed state; optionally add a strict helper variant that reverts on invalid tick ordering/bounds and zero owner.

INFO

`StateLibrary.getSlot0` exposes packed protocol-fee lanes without direction-aware return/docs

Locations:

`src/libraries/StateLibrary.sol:31-39`

`src/libraries/StateLibrary.sol:53-61`

`src/types/Slot0.sol:9-22`

`src/libraries/ProtocolFeeLibrary.sol:17-23`

`src/libraries/Pool.sol:283-285`

`StateLibrary.getSlot0` returns `uint24 protocolFee` as the raw 24-bit field from slot0 (`and(shr(184, data), 0xFFFFFF)`), but this field is bi-directional packed data (upper 12 bits = 1->0 fee, lower 12 bits = 0->1 fee) per `Slot0` layout.

This means consumers must decode by swap direction before using it as a fee scalar. Directional decoding helpers exist (`getZeroForOneFee`, `getOneForZeroFee`), and core swap logic does this correctly in `Pool.swap`.

Concrete simulation of the stated footgun:

- Packed fee set to `(900 << 12) | 100 = 3,686,500` (valid packed config).
- For a `oneForZero` quote, correct protocol fee is `900`, not `3,686,500`.
- With `lpFee=3000`, correct swap fee is `3,898` pips; misusing raw packed value gives `3,678,441` pips.

So the risk is integration misuse (misquotes/misaccounting in downstream contracts or off-chain quoting), not v4-core execution failure. Core accounting/state remain correct because swap execution decodes direction-specific fees internally.

INFO

`StateLibrary.getTickBitmap` parameter is mis-named as `tick` but is actually `wordPos`

Locations:

`src/libraries/StateLibrary.sol:190-213`

`src/libraries/Pool.sol:85-87`

`src/libraries/TickBitmap.sol:31-40`

`StateLibrary.getTickBitmap` accepts an `int16` argument named `tick`, but the underlying pool storage key is `tickBitmap[wordPos]` (`mapping(int16 wordPos => uint256)`).

```
function getTickBitmap(IPoolManager manager, PoolId poolId, int16 tick)
...
bytes32 slot = keccak256(abi.encodePacked(int256(tick), tickBitmapMapping));
```

This slot derivation is correct for `wordPos`, not an `int24` price tick. The naming/NatSpec can mislead integrators into passing a raw tick cast to `int16`.

Concrete example (`tickSpacing=60`, `tickUpper=887220`):

- Correct: `compressed = 887220/60 = 14787`, `wordPos = 14787 >> 8 = 57`, `bitPos=195`, so read `tickBitmap[57]`.
- Misuse: `int16(887220) = -30284`, so helper reads `tickBitmap[-30284]` (different word).

This can return unrelated bitmap data (or zero), causing incorrect caller-side tick traversal/quotes in integrations that misuse the API.

Impact:
Limited to integrator/off-chain/on-chain consumer behavior that calls this helper incorrectly. It does **not** alter v4-core pool accounting or funds because this is a read-only helper.

Recommendation:

Rename parameter/NatSpec to `wordPos` and explicitly document expected derivation (`TickBitmap.position(compress(tick, tickSpacing)).wordPos` or equivalent). Optionally add a helper that accepts `(tick, tickSpacing)` and derives `wordPos` internally to reduce misuse risk.

INFO

Swap event reports pre-afterSwap pool delta; caller net delta can differ when hooks return deltas

Locations:

`src/PoolManager.sol:219-225`

`src/PoolManager.sol:239-249`

`src/libraries/Hooks.sol:283-314`

`src/PoolManager.sol:173-177`

`src/libraries/Hooks.sol:207-244`

`PoolManager` emits `Swap` inside `_swap` **before** invoking `key.hooks.afterSwap(...)`. When the pool's hook has `AFTER_SWAP_RETURNS_DELTA_FLAG`, `Hooks.afterSwap` computes a `hookDelta` and applies:

```
swapDelta = swapDelta - hookDelta;
```

So the emitted `Swap.amount0/amount1` are the pre-after-hook pool delta, while the caller's final accounted delta can be different.

Concrete trace (from the `FeeTakingHook` path): for exact-in 1000, pre-hook delta is `(-1000, +998)`, hook returns `+12` unspecified, final caller delta becomes `(-1000, +986)`. The hook receives 12, caller receives 986, but the `Swap` event amount1 is still 998.

For `modifyLiquidity`, the event is also emitted before `afterModifyLiquidity`, and after-hooks can alter `callerDelta` via `callerDelta = callerDelta - hookDelta`. Note however that `ModifyLiquidity` event does not contain token amount deltas at all (only position/liquidity metadata), so it cannot encode caller net token movements either way.

Impact:

No on-chain fund loss or accounting break was found. This is an off-chain interpretation risk: consumers that treat swap event amounts as the caller's net settled amounts (instead of pool deltas) may mis-account on hook-enabled pools.

`IPoolManager` docs already describe `Swap.amount0/amount1` as pool balance deltas and warn that hooks can alter returned `swapDelta`, so this is best classified as informational semantics/consumer-footgun risk.

Recommendation:

Document this behavior prominently for integrators/indexers:

- `Swap` event amounts are pre-after-hook pool deltas.
- caller net amounts should come from returned deltas / settlement traces, not from `Swap.amount0/amount1` alone on delta-returning hook pools.
- `ModifyLiquidity` event should not be used to infer token in/out amounts.

INFO

`computeSwapStep` NatSpec misstates fee-bound source; unchecked subtraction would wrap for invalid callers

Locations:

`src/libraries/SwapMath.sol:50-66`

`src/libraries/SwapMath.sol:100-104`

`src/libraries/Pool.sol:300-305`

`src/libraries/ProtocolFeeLibrary.sol:34-45`

`SwapMath.computeSwapStep` documents that `feePips <= MAX_SWAP_FEE` is ensured "using `LPFeeLibrary.isValid`", but the actual runtime input is the **total swap fee** (`swapFee`, LP+protocol) computed in `Pool.swap`.

- In `Pool.swap`, `computeSwapStep(..., swapFee)` is called, where `swapFee = protocolFee == 0 ? lpFee : calculateSwapFee(protocolFee, lpFee)`.
- Therefore, the bound does not come solely from LP fee validation; it comes from the combination of:

1) LP fee validation (`validate` / `removeOverrideFlagAndValidate`), 2) protocol fee validation (`isValidProtocolFee`), and 3) `calculateSwapFee` composition.

Because `computeSwapStep` is in an `unchecked` block and uses `MAX_SWAP_FEE - feePips`, an out-of-range caller would get wraparound behavior:

- exact-in path can revert in `FullMath.mulDiv` for realistic sizes (e.g., abs amount `1e18`, `feePips=1_000_001`),
- exact-out path can silently produce nonsense fee charging (e.g., fee rounding to `1` wei due enormous wrapped denominator).

Current v4-core call paths appear safe because production constraints keep `swapFee <= 1_000_000` and exact-out is blocked when `swapFee >= MAX_SWAP_FEE`, so this is primarily a documentation/maintainability issue rather than a present exploit.

INFO

Boundary-state tick/price divergence can route `donate()` to tick-side LPs rather than `getTick-AtSqrtPrice(sqrtPrice)` side

Locations:

`src/libraries/Pool.sol:406-409`

`src/libraries/Pool.sol:410-429`

`src/libraries/Pool.sol:463-475`

`src/libraries/Pool.sol:492-506`

`src/interfaces/IPoolManager.sol:167-170`

`Pool.swap` can intentionally persist a boundary state where `slot0.sqrtPriceX96` is exactly at tick `n` while `slot0.tick` is `n-1` for `zeroForOne` steps that end exactly at `step.sqrtPriceNextX96`.

```
if (result.sqrtPriceX96 == step.sqrtPriceNextX96) {
 ...
 result.tick = zeroForOne ? step.tickNext - 1 : step.tickNext;
}
```

`donate()` does not derive eligibility from sqrt price; it adds fee growth against current `state.liquidity`, and position fee attribution uses `tickCurrent = self.slot0.tick()` in `getFeeGrowthInside`.

```
uint128 liquidity = state.liquidity;
state.feeGrowthGlobal0X128 += amount0 * Q128 / liquidity;
...
int24 tickCurrent = self.slot0.tick();
```

So in boundary state (`sqrtPriceX96 = getSqrtPriceAtTick(n)`, `slot0.tick = n-1`), donation accrues to the tick-side associated with `n-1` (left side), not the side an integration may infer from `getTickAtSqrtPrice(sqrtPriceX96)=n`.

Concrete trace: with ranges `[-60,0]` and `[0,60]`, initialize around tick 30, zero-ForOne swap to `sqrtPriceLimit = getSqrtPriceAtTick(0)` and stop exactly at boundary. Final state can be `sqrtPrice@0` with `tick=-1`; immediate donate then credits the `[-60,0]` side for that donation event.

Impact calibration:

This is a real behavior but primarily an integration footgun. Core code and interface docs explicitly warn about this edge case, including that `donate` uses `slot0.tick` and may differ from `getTickAtSqrtPrice(slot0.sqrtPriceX96)`.

Recommendation:

Any incentive/hook/offchain donation trigger that targets a price boundary should validate both `slot0.tick` and `slot0.sqrtPriceX96` (or rely on tick semantics explicitly) before donating.

INFO

`getTickAtSqrtPrice` uses unchecked `int24` narrowing casts and relies on fixed numeric invariants

Locations:

`src/libraries/TickMath.sol:127-129`

`src/libraries/TickMath.sol:228-235`

`TickMath.getTickAtSqrtPrice()` computes candidate ticks via narrowing casts:

```
int24 tickLow = int24((log_sqrt10001 - 3402992956809132418596140100660247210) >> 128);
int24 tickHi  = int24((log_sqrt10001 + 291339464771989622907027621153398088495) >> 128);
```

There is no explicit post-cast range assertion before `tickHi` is used in `getSqrtPriceAtTick(tickHi)`.

For the current deployed constants and input guard (`MIN_SQRT_PRICE <= sqrtPriceX96 < MAX_SQRT_PRICE`), this is safe in practice: sampled boundary and randomized traces showed pre-cast values remain around protocol ticks (observed `preLow [-887273, 887271]`, `preHi [-887272, 887272]`), so casts do not truncate/wrap.

However, correctness depends on these hard-coded constants and approximation assumptions remaining aligned. If constants are modified in a future version, cast wrapping can occur and may lead to either:

- local revert via `getSqrtPriceAtTick(tickHi)` (out-of-range wrapped tick), or
- silent wrong tick selection (if wrapped residue lands inside `int24`).

Because this is not attacker-triggerable in current code and requires code/constant drift, this is an informational robustness issue (defensive hardening), not an exploitable production vulnerability.

INFO

TickMath usable-tick helpers do not guard `tickSpacing` and will panic on zero input

Location:

`src/libraries/TickMath.sol:39-49`

`TickMath.maxUsableTick()` and `TickMath.minUsableTick()` divide by `tickSpacing` directly without validating the input.

```
return (MAX_TICK / tickSpacing) * tickSpacing;
return (MIN_TICK / tickSpacing) * tickSpacing;
```

With concrete values:

- `tickSpacing = 0` causes a Solidity division-by-zero panic.
- `tickSpacing = -3` returns `max=887271`, `min=-887271` (i.e., behavior depends on signed division semantics rather than an explicit domain check).

This is a real behavior in the helper functions, but in **v4-core runtime paths** it is not currently exploitable: `PoolManager.initialize` enforces `1 <= tickSpacing <= type(int16).max` before pool creation, and these helpers are not used by production `src/` call paths (only test code uses them).

So the issue is best framed as a defensive API hardening/documentation gap for future or external consumers that might pass unvalidated user input into these helpers.

INFO

PoolManager.unlock() missing noDelegateCall modifier enables delegatecall entrypoint (inconsistent execution-flow hardening)

Locations:

`src/PoolManager.sol:103-114`

`src/NoDelegateCall.sol:28-32`

Description:

`PoolManager` inherits `NoDelegateCall` and applies `noDelegateCall` to most state-changing external entrypoints (e.g., `modifyLiquidity`, `swap`, `donate`). However, `unlock()` is **not** protected by `noDelegateCall`, unlike the test `ProxyPoolManager` implementation.

While this does not let an attacker mutate the real `PoolManager` storage via delegatecall (delegatecall writes to the caller's context), it breaks the intended invariant that `PoolManager` methods cannot be executed via delegatecall, and it creates a surprising execution path where transient lock state (`Lock.unlock()`/`Lock.lock()`) and callback dispatch (`IUnlockCallback(msg.sender).unlockCallback`) occur in the caller's context.

Evidence:

```
// src/PoolManager.sol
function unlock(bytes calldata data) external override returns (bytes memory result) {
 if (Lock.isUnlocked()) AlreadyUnlocked.selector.revertWith();
 Lock.unlock();
 result = IUnlockCallback(msg.sender).unlockCallback(data);
 if (NonzeroDeltaCount.read() != 0) CurrencyNotSettled.selector.revertWith();
 Lock.lock();
}
```

No `noDelegateCall` modifier is present.

Impact:

- Allows unexpected delegatecall-based execution of the unlock/callback flow in a different contract context.
- Weakens the protocol's execution-flow assumptions and hardening against proxy/delegatecall misuse.

Recommendation:
Apply `noDelegateCall` consistently to `unlock()` as with other external entrypoints (unless delegatecall support is explicitly intended and documented).

INFO

ProtocolFeeLibrary.calculateSwapFee lacks bounds validation on lpFee/self; invalid inputs can yield swapFee > 100%

Locations:

`src/libraries/ProtocolFeeLibrary.sol:34-46`

`src/libraries/LPFeeLibrary.sol:34-78`

`src/libraries/Pool.sol:297-313`

`src/libraries/SwapMath.sol:63-75`

Description:

`ProtocolFeeLibrary.calculateSwapFee(uint16 self, uint24 lpFee)` claims (via comments) that "The swap fee is capped at 100%", but the function itself **does not enforce** `lpFee <= 1_000_000` (or `self <= MAX_PROTOCOL_FEE`). Instead it simply masks the inputs (`self &= 0xfff`, `lpFee &= 0xffffff`).

If a caller ever passes an unvalidated `lpFee` (e.g., including hook flag bits like `OVERRIDE_FEE_FLAG`, or any value > `LPFeeLibrary.MAX_LP_FEE`), `swapFee` can exceed `SwapMath.MAX_SWAP_FEE` (1e6).

This is dangerous because `SwapMath.computeSwapStep` assumes `feePips <= MAX_SWAP_FEE` and uses `MAX_SWAP_FEE - feePips` in unchecked arithmetic. A `swapFee > MAX_SWAP_FEE` can cause underflow and severely incorrect swap math.

In the current core flow, `Pool.swap` appears to call `removeOverrideFlagAndValidate()` before passing `lpFee` into `calculateSwapFee`, so this may be a **latent footgun** rather than an immediately exploitable bug. However, the lack of internal validation makes the library easy to misuse and violates the stated cap unless callers strictly validate.

Evidence:

```
function calculateSwapFee(uint16 self, uint24 lpFee) internal pure returns (uint24 swapFee) {
 assembly ("memory-safe") {
 self := and(self, 0xfff)
 lpFee := and(lpFee, 0xffffff)
 let numerator := mul(self, lpFee)
 swapFee := sub(add(self, lpFee), div(numerator, PIPS_DENOMINATOR))
 }
}
```

Impact:

If any path (now or in future refactors) calls `calculateSwapFee` with an unvalidated `lpFee`, swaps can compute with an effective fee > 100%, triggering underflow in swap step fee calculations and leading to incorrect amounts (potentially value loss or invariant breaks).

Recommendation:

Add explicit bounds checks (or at least assert assumptions) inside `calculateSwapFee`, or narrow the accepted types/encoding so invalid fees cannot reach this function.

INFO

SqrtPriceMath.getNextSqrtPriceFromAmount0RoundingUp can return 0 when liquidity=0 (missing validation in helper), risking incorrect price if misused

Locations:

`src/libraries/SqrtPriceMath.sol:20-74`

`src/libraries/SqrtPriceMath.sol:31-74`

Description:

`SqrtPriceMath.getNextSqrtPriceFromAmount0RoundingUp` is a low-level helper used by the validated wrappers (`getNextSqrtPriceFromInput`/`getNextSqrtPriceFromOutput`). The wrappers explicitly revert when `sqrtPX96 == 0` or `liquidity == 0`.

However, the helper itself does **not** validate `liquidity > 0` (or `sqrtPX96 > 0`) and, in the `add == true` branch, can return `0` when `liquidity == 0`:

- `numerator1 = uint256(liquidity) << 96` becomes `0`
- either path returns `mulDivRoundingUp(0, ...)` or `divRoundingUp(0, ...)` ' `0`

A returned sqrt price of `0` is not a meaningful/valid pool price and can propagate into higher-level logic as a business-logic error if this helper is called directly (now or in future refactors), rather than via the validated wrappers.

Evidence:

```
function getNextSqrtPriceFromAmount0RoundingUp(uint160 sqrtPX96, uint128 liquidity, uint256
amount, bool add)
 internal pure returns (uint160)
{
 if (amount == 0) return sqrtPX96;
 uint256 numerator1 = uint256(liquidity) << FixedPoint96.RESOLUTION;

 if (add) {
 ...
 return uint160(UnsafeMath.divRoundingUp(numerator1, (numerator1 / sqrtPX96) + amount));
 } else {
 ... // remove branch reverts on underflow/overflow
 }
}
```

Impact:

Currently, core calls this helper only via wrappers that enforce `liquidity != 0`, so the impact is mainly a **footgun**:

- direct/internal use elsewhere could silently compute a next price of `0`, leading to incorrect swap/position math and potential downstream reverts or mis-accounting.

Recommendation:

Add explicit input validation (`liquidity > 0` and `sqrtPX96 > 0`) in the helper itself, or clearly document that it must only be called through the validated wrappers.

INFO

TickMath.getTickAtSqrtPrice rejects MAX_SQRT_PRICE even though it is documented as getSqrtPriceAtTick(MAX_TICK) output

Locations:

`src/libraries/TickMath.sol:30-37`

`src/libraries/TickMath.sol:116-129`

`src/libraries/TickMath.sol:7-17`

`src/libraries/TickMath.sol:116-130`

`src/libraries/TickMath.sol:30-36`

Description:

`TickMath` documents `MAX_SQRT_PRICE` as the maximum value returned by `getSqrtPriceAtTick` and "Equivalent to getSqrtPriceAtTick(MAX_TICK)".

However, `getTickAtSqrtPrice` explicitly rejects `sqrtPriceX96 >= MAX_SQRT_PRICE` (strict upper bound). This means the round-trip identity does not hold at the upper boundary:

- `getSqrtPriceAtTick(MAX_TICK)` can return `MAX_SQRT_PRICE`,
- but `getTickAtSqrtPrice(MAX_SQRT_PRICE)` reverts.

Even if this is inherited from historical Uniswap behavior, it is a business-logic/documentation mismatch that can surprise integrators and lead to avoidable reverts or off-by-one boundary errors.

Evidence:

```
uint160 internal constant MAX_SQRT_PRICE = 1461446703...70342;

// Equivalent: if (sqrtPriceX96 < MIN_SQRT_PRICE || sqrtPriceX96 >= MAX_SQRT_PRICE) revert
if ((sqrtPriceX96 - MIN_SQRT_PRICE) > MAX_SQRT_PRICE_MINUS_MIN_SQRT_PRICE_MINUS_ONE) {
 InvalidSqrtPrice.selector.revertWith(sqrtPriceX96);
}
```

Impact:

Consumer contracts that assume `getTickAtSqrtPrice(getSqrtPriceAtTick(tick))` always succeeds for all valid ticks can unexpectedly revert at the upper boundary.

Recommendation:
Align docs with actual domain restrictions (explicitly state `getTickAtSqrtPrice` requires `sqrtPriceX96 < MAX_SQRT_PRICE`).

Advisory to Clients

Cecuro highly recommends scheduling a follow-up security assessment within six months of this report or immediately after any significant modifications to the codebase, whichever occurs first. This practice is essential for preserving the project's security posture and identifying potential vulnerabilities that may arise from code changes or updates.

Disclaimer

The smart contracts submitted for analysis have been reviewed using Cecuro.ai's AI security analysis system, applying industry-standard vulnerability detection patterns and best practices at the time of this report. The findings disclosed in this report reflect the AI system's assessment of the submitted source code.

This report contains no statements or warranties on the identification of all vulnerabilities or the overall security of the code. Any security review methodology may not detect all potential issues, particularly novel attack vectors, complex business logic flaws, or economic exploits. The report covers the code submitted at the specified version and may not be relevant after any modifications.

Do not consider this report as a final and sufficient assessment regarding the security, utility, or bug-free status of the code. We recommend complementing this analysis with additional independent audits and a public bug bounty program to strengthen the security posture of your smart contracts.

Smart contracts are deployed and executed on blockchain platforms. The platform, its programming language, compiler, and other related software may contain vulnerabilities beyond the scope of code-level analysis. Cecuro.ai cannot guarantee the explicit security of the analyzed smart contracts.

This report does not constitute financial, legal, or investment advice. It is not a recommendation to buy, sell, or invest in any token or project, nor an endorsement of any kind. Users should conduct their own independent due diligence.

cecuro

Agentic Smart Contract Auditing