cecuro

Audit Report

March 2, 2026

PROJECT

Uniswap

Audit Overview

Project:	Uniswap
Repository:	https://github.com/Cecuro/uniswap-v3-core
Audit Date:	March 2, 2026
Commit:	d8b1c635
Scope:	35 files

0	0	0	1	51
CRITICAL	HIGH	MEDIUM	LOW	INFO

Security audit completed.

Audit Scope

The following 35 files were included in this security audit:

- contracts/NoDelegateCall.sol
- contracts/UniswapV3Factory.sol
- contracts/UniswapV3Pool.sol
- contracts/UniswapV3PoolDeployer.sol
- contracts/interfaces/IERC20Minimal.sol
- contracts/interfaces/IUniswapV3Factory.sol
- contracts/interfaces/IUniswapV3Pool.sol
- contracts/interfaces/IUniswapV3PoolDeployer.sol
- contracts/interfaces/callback/IUniswapV3FlashCallback.sol
- contracts/interfaces/callback/IUniswapV3MintCallback.sol
- contracts/interfaces/callback/IUniswapV3SwapCallback.sol
- contracts/interfaces/pool/IUniswapV3PoolActions.sol
- contracts/interfaces/pool/IUniswapV3PoolDerivedState.sol
- contracts/interfaces/pool/IUniswapV3PoolEvents.sol
- contracts/interfaces/pool/IUniswapV3PoolImmutables.sol
- contracts/interfaces/pool/IUniswapV3PoolOwnerActions.sol
- contracts/interfaces/pool/IUniswapV3PoolState.sol
- contracts/libraries/BitMath.sol
- contracts/libraries/FixedPoint128.sol
- contracts/libraries/FixedPoint96.sol
- contracts/libraries/FullMath.sol
- contracts/libraries/LICENSE
- contracts/libraries/LICENSE_MIT
- contracts/libraries/LiquidityMath.sol
- contracts/libraries/LowGasSafeMath.sol
- contracts/libraries/Oracle.sol
- contracts/libraries/Position.sol
- contracts/libraries/SafeCast.sol
- contracts/libraries/SqrtPriceMath.sol
- contracts/libraries/SwapMath.sol
- contracts/libraries/Tick.sol
- contracts/libraries/TickBitmap.sol
- contracts/libraries/TickMath.sol
- contracts/libraries/TransferHelper.sol
- contracts/libraries/UnsafeMath.sol

Findings

LOW

Overly permissive Solidity pragmas are incompatible with used language features and arithmetic assumptions

Locations:

`contracts/libraries/TickMath.sol:2`

`contracts/libraries/TickMath.sol:48`

`contracts/libraries/Tick.sol:2`

`contracts/libraries/Tick.sol:48`

`contracts/libraries/SqrtPriceMath.sol:2`

`contracts/libraries/SqrtPriceMath.sol:79-90`

`contracts/libraries/FullMath.sol:2`

`contracts/libraries/FullMath.sol:120`

Description:

Several scope libraries declare compiler pragmas that suggest compatibility with older Solidity versions, but the implementations use `type(T).max` / `type(T).min`, which requires Solidity **>= 0.6.8**. In addition, some math patterns rely on **pre-0.8 unchecked arithmetic** (wrapping) semantics.

This is a business-logic risk because downstream users/forks/integrations may select a compiler version that appears allowed by the pragma but either:

- **fails to compile** (older solc), or
- compiles after pragma edits but changes arithmetic semantics (e.g., moving toward 0.8 without auditing overflow/underflow behavior), potentially breaking swap/price/fee math.

Evidence:

- `TickMath.sol` declares `pragma solidity >=0.5.0 <0.8.0;` but uses `type(uint256).max`:

```
if (tick > 0) ratio = type(uint256).max / ratio;
```

- `Tick.sol` declares `pragma solidity >=0.5.0 <0.8.0;` but uses `type(uint128).max`:

```
return type(uint128).max / numTicks;
```

- `SqrtPriceMath.sol` declares `pragma solidity >=0.5.0;` but uses `type(uint160).max`:

```
amount <= type(uint160).max ? ... : ...
```

- `FullMath.sol` declares `pragma solidity >=0.4.0 <0.8.0;` but uses `type(uint256).max`:

```
require(result < type(uint256).max);
```

Impact:

- Misleading version ranges can cause integrators to believe a compiler is supported when it is not, or to "fix" pragmas in a fork and inadvertently change arithmetic semantics, leading to silent economic/accounting deviations or unexpected reverts in core math.

Recommendation:

Tighten pragmas (or add explicit documentation) to the **true supported compiler range** and explicitly document reliance on unchecked arithmetic where applicable.

INFO

TransferHelper.safeTransfer vulnerable to returndata-bomb gas/memory DoS

Locations:

`contracts/libraries/TransferHelper.sol:14-22`

`contracts/UniswapV3Pool.sol:503-510`

`contracts/UniswapV3Pool.sol:771-784`

`contracts/UniswapV3Pool.sol:791-809`

`contracts/UniswapV3Pool.sol:856-865`

Description:

`TransferHelper.safeTransfer` uses a high-level low-level call that **copies the full returndata into memory** (`bytes memory data`). A malicious or upgradeable ERC-20 can return an extremely large returndata payload ("returndata bomb"), forcing massive memory expansion and/or copy costs, causing the call to run out of gas and revert.

This is a **computational-bounds / resource-exhaustion** issue: the gas cost is controlled by the token contract (and indirectly by the caller choosing tokens), not by the pool.

Vulnerable Code:

`contracts/libraries/TransferHelper.sol:14-22`

```
(bool success, bytes memory data) =
 token.call(abi.encodeWithSelector(IERC20Minimal.transfer.selector, to, value));
require(success && (data.length == 0 || abi.decode(data, (bool))), 'TF');
```

Why this matters in-core:

The pool uses `TransferHelper.safeTransfer` in multiple core state transitions (examples):

- `collect()` pays out fees to LPs (`UniswapV3Pool.sol:503-510`)
- `swap()` pays output to `recipient` (`UniswapV3Pool.sol:771-784`)
- `flash()` transfers the flash amounts to `recipient` (-`UniswapV3Pool.sol:805-808`)

- `collectProtocol()` pays protocol fees (`UniswapV3Pool.sol:856-865`)

Any of these can be made to OOG-revert if `token0` or `token1` returns very large returndata on `transfer`, effectively **bricking** these operations for that pool.

Impact:

- **High impact DoS (per affected pool):** swaps, fee collection, protocol fee collection, and flash payouts can become unexecutable within block gas limits.
- If the token is upgradeable, the pool can be bricked after deployment by upgrading token logic to returnbomb.

Recommendation:

Avoid copying unbounded returndata for ERC-20 calls. Use an assembly pattern that:

- performs the call,
- only inspects `returndatasize` up to 32 bytes (or strictly enforces 0/32-byte return),
- does not allocate/copy arbitrary-length returndata into memory.

INFO

Callback implementers can be fooled if they authenticate pools by trusting `pool.factory()` / immutables (pool constructor trusts unverified deployer-supplied parameters)

Locations:

```
contracts/interfaces/IUniswapV3PoolDeployer.sol:8-25
contracts/interfaces/callback/IUniswapV3SwapCallback.sol:7-15
contracts/interfaces/callback/IUniswapV3MintCallback.sol:7-12
contracts/interfaces/callback/IUniswapV3FlashCallback.sol:7-12
contracts/UniswapV3Pool.sol:117-123
```

Description:

A common (but unsafe) pattern for callback implementers is to authenticate the pool by reading pool immutables like `factory()`, `token0()`, `token1()`, or `fee()`. This is risky because the pool contract's constructor **blindly trusts** `IUniswapV3PoolDeployer(msg.sender).parameters()`.

An attacker can deploy an *arbitrary* contract implementing `parameters()` and then deploy a `UniswapV3Pool` instance whose immutables claim:

- `factory == canonicalFactory`
- `token0/token1/fee/tickSpacing` chosen by attacker

Such a pool was **not deployed by the canonical factory**, but naive authentication like `require(IUniswapV3Pool(msg.sender).factory() == canonicalFactory)` will pass.

The callback interface docs say: > "The caller of this method must be checked to be a UniswapV3Pool deployed by the canonical UniswapV3Factory."

…but they do not warn that checking the pool's reported `factory()` is insufficient.

Evidence:

Pool constructor trusts deployer parameters:

```
(factory, token0, token1, fee, _tickSpacing) = IUniswapV3PoolDeployer(msg.sender).parameters-
();
```

Impact:

Downstream callback implementers can be drained if they:

- treat `pool.factory()` (or other immutables) as authoritative, instead of
- computing/verifying the pool address via canonical factory's `getPool()` or CREATE2 derivation.

Recommendation:

Explicitly document that `pool.factory()` and other immutables are **not sufficient** for authentication; implementers must validate that `msg.sender` equals the canonical factory's registered pool for the given `(token0, token1, fee)` (or verify via CREATE2 + init code hash).

INFO

Core contracts are not proxy/upgrade-safe: constructor-only initialization and immutables will brick or misconfigure proxy deployments

Locations:

`contracts/UniswapV3Factory.sol:22-32`

`contracts/UniswapV3Factory.sol:34-72`

`contracts/UniswapV3Pool.sol:41-55`

`contracts/UniswapV3Pool.sol:117-123`

`contracts/NoDelegateCall.sol:8-26`

Description:

The Uniswap V3 core contracts in scope are **not compatible with upgradeable proxy patterns**, but there are **no explicit guardrails preventing a proxy deployment**. If a deployer/integrator attempts to make these contracts upgradeable by placing them behind a proxy (transparent/UUPS/minimal proxy), critical initialization performed in constructors will not run, and the pool's immutable configuration will not be proxy-specific.

This results in **bricked deployments** (e.g., factory owner remains `address(0)` and fee tiers are never enabled) and/or **misconfigured pools** (immutables resolve to the implementation's embedded constants under `DELEGATECALL`, not the proxy's desired per-pool configuration).

Evidence:

Factory relies on constructor for essential initialization:

`UniswapV3Factory` sets `owner` and enables initial fee tiers only in the constructor:

```
constructor() {
 owner = msg.sender;
 ...
 feeAmountTickSpacing[500] = 10;
 ...
}
```

(contracts/UniswapV3Factory.sol:22-32)

If used via a proxy, the constructor does not run, so:

- `owner` stays `address(0)`
- `feeAmountTickSpacing[...]` stays 0 for all fees

This bricks core functions:

```
int24 tickSpacing = feeAmountTickSpacing[fee];
require(tickSpacing != 0);
```

(contracts/UniswapV3Factory.sol:43-45)

and prevents recovery because enabling fees is owner-gated:

```
require(msg.sender == owner);
```

(contracts/UniswapV3Factory.sol:61-63)

Pool uses immutables and constructor-time parameterization:

The pool's critical configuration is immutable:

```
address public immutable override factory;
address public immutable override token0;
address public immutable override token1;
uint24 public immutable override fee;
int24 public immutable override tickSpacing;
uint128 public immutable override maxLiquidityPerTick;
```

(contracts/UniswapV3Pool.sol:41-55)

and is set only in the constructor by reading deployer parameters:

```
(factory, token0, token1, fee, _tickSpacing) = IUniswapV3PoolDeployer(msg.sender).parameters-
();
```

(contracts/UniswapV3Pool.sol:117-123)

Under proxy `DELEGATECALL`, immutables resolve to values embedded in the *implementation bytecode*, so a single implementation cannot safely serve multiple proxy instances with different (token0, token1, fee, tickSpacing).

NoDelegateCall partially blocks proxy usage but doesn't prevent it at deployment time:

`NoDelegateCall` causes `noDelegateCall`-guarded methods to revert when invoked via proxy delegatecall:

```
require(address(this) == original);
```

(contracts/NoDelegateCall.sol:18-20)

This makes proxy deployments fail in surprising ways (some methods revert, others may still execute), rather than failing fast at deployment.

Impact:

High operational risk / protocol bricking in any deployment that mistakenly uses proxy patterns:

- Factory owner and fee tiers not initialized ' pools cannot be created and fees cannot be enabled.
- Pool logic cannot be safely upgraded via proxy because immutables are fixed in the implementation.
- Partial `noDelegateCall` coverage can lead to confusing "partially working" proxies, increasing the chance of misconfiguration and stuck assets in integrator wrappers.

Recommendation:

- Explicitly document that these contracts are **not upgradeable** and must not be used behind proxies.
- Consider adding hard anti-proxy guardrails (e.g., deployment-time checks or consistently applying `noDelegateCall` to *all* external entrypoints) if you want to fail closed under delegatecall/proxy environments.

INFO

Factory can enable fee tiers with extremely small tickSpacing, increasing worst-case swap steps and making pools impractical under gas limits

Locations:

`contracts/UniswapV3Factory.sol:61-68`

`contracts/UniswapV3Pool.sol:640-731`

`contracts/libraries/TickBitmap.sol:34-76`

Description:

`UniswapV3Factory.enableFeeAmount()` allows the factory owner to configure `tickSpacing` as low as `1` (it only requires `tickSpacing > 0 && tickSpacing < 16384`).

Very small `tickSpacing` increases the number of discrete price boundaries that can be traversed during a swap, which increases the number of iterations in `UniswapV3Pool.swap()`'s step loop. Even if no ticks are initialized, `tickBitmap.nextInitializedTickWithinOneWord()` advances at most one bitmap word per step (256 * tickSpacing ticks), so worst-case steps scale roughly with `priceRange / (256 * tickSpacing)`.

As a result, a fee tier misconfigured with a very small `tickSpacing` can create pools where large (or adversarially constructed) swaps routinely exceed gas limits and revert.

Evidence:

Factory permits small tickSpacing:

```
require(tickSpacing > 0 && tickSpacing < 16384);
```

(contracts/UniswapV3Factory.sol:67)

Swap loop iterates per step:

```
while (state.amountSpecifiedRemaining != 0 && state.sqrtPriceX96 != sqrtPriceLimitX96) {
 (step.tickNext, step.initialized) = tickBitmap.nextInitializedTickWithinOneWord(
 state.tick,
 tickSpacing,
 zeroForOne
 );
 ...
}
```

(contracts/UniswapV3Pool.sol:641-650)

Tick bitmap search is bounded to one word (256 bits) from the starting point:

```
/// @return next The next initialized or uninitialized tick up to 256 ticks away from the
current tick
function nextInitializedTickWithinOneWord(...)
```

(contracts/libraries/TickBitmap.sol:34-76)

Impact:

- **Configuration-driven computational DoS:** a newly enabled fee tier (and pools created under it) can be hard/impossible to trade through for swaps that need to move across many steps, leading to frequent OOG reverts.
- This risk is amplified when LPs also initialize many ticks (dense liquidity), since smaller `tickSpacing` increases the maximum possible tick density.

Recommendation:

- Consider enforcing a **minimum tickSpacing** (or tier-specific minimums) based on gas budget assumptions.
- If flexibility is required, document that fee tiers with very small `tickSpacing` can make swaps impractical and should be avoided on gas-constrained chains.

INFO

Factory createPool() does not validate token addresses are contracts, allowing creation of pools that cannot function (EOA/non-ERC20 tokens)

Locations:

`contracts/UniswapV3Factory.sol:35-51`

`contracts/UniswapV3Pool.sol:137-155`

Description:

`UniswapV3Factory.createPool()` validates that tokens are non-zero and distinct, but it does **not** validate that `token0`/`token1` are contracts or implement the expected ERC-20 interface.

As a result, anyone can create a pool for arbitrary addresses (including EOAs). Such pools will later revert on core operations because the pool assumes the token addresses respond to `balanceOf()`.

This is an input-validation gap that can lead to creation of unusable pools and can confuse users/integrators relying on `PoolCreated` events.

Evidence:

Factory does not check `extcodesize(token)`:

```
require(tokenA != tokenB);
(address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
require(token0 != address(0));
...
pool = deploy(address(this), token0, token1, fee, tickSpacing);
emit PoolCreated(token0, token1, fee, tickSpacing, pool);
```

Pool requires `balanceOf()` to succeed and return at least 32 bytes:

```
(bool success, bytes memory data) =
 token0.staticcall(abi.encodeWithSelector(IERC20Minimal.balanceOf.selector, address(this-
)));
require(success && data.length >= 32);
```

Impact:

- Users can create pools that are permanently unusable (core functions revert due to token calls returning empty data).
- Event-driven indexers/UIs that do not validate token contracts can surface misleading pools, increasing phishing/confusion risk.

Recommendation:

If unusable pools are not desired, validate that `token0` and `token1` have code (`extcodesize > 0`) and/or perform an ERC-20 interface sanity check. If permissionless "predeploy token addresses" is intended, explicitly document this risk for indexers/UIs.

INFO

Factory ownership can be irreversibly bricked by setting owner to address(0), permanently disabling protocol-fee administration/collection

Locations:

`contracts/UniswapV3Factory.sol:54-58`

`contracts/UniswapV3Pool.sol:111-115`

`contracts/UniswapV3Pool.sol:836-868`

Description:

`UniswapV3Factory.setOwner()` allows the current owner to set the factory owner to `address(0)` (or any non-functional address) without any validation.

Because Uniswap V3 pools gate protocol-fee administration and collection via `onlyFactoryOwner` (which checks `msg.sender == IUniswapV3Factory(factory).owner()`), setting the factory owner to `address(0)` permanently disables:

- `UniswapV3Factory.enableFeeAmount()` and future `setOwner()` calls
- `UniswapV3Pool.setFeeProtocol()` on all pools
- `UniswapV3Pool.collectProtocol()` on all pools

If protocol fees have been enabled (non-zero `slot0.feeProtocol`) at any point, renouncing/bricking ownership can also:

- Permanently lock accumulated protocol fees in `protocolFees` inside each pool
- Leave protocol fees enabled indefinitely (cannot be turned off), reducing LP fee share while the protocol share becomes uncollectable

This is an access-control footgun: a single mistaken transaction can cause permanent loss of privileged control and permanent loss/lock of protocol-owned fees.

Vulnerable Code:

Factory owner can be set to any address (including 0):

```
function setOwner(address _owner) external override {
 require(msg.sender == owner);
 emit OwnerChanged(owner, _owner);
 owner = _owner;
}
```

Pools rely on factory owner for privileged operations:

```
modifier onlyFactoryOwner() {
 require(msg.sender == IUniswapV3Factory(factory).owner());
 _;
}

function setFeeProtocol(...) external override lock onlyFactoryOwner { ... }
function collectProtocol(...) external override lock onlyFactoryOwner returns (...) { ... }
```

Impact:

High impact on protocol governance and protocol-fee funds:

- Permanent denial-of-service of all owner-only methods
- Potential permanent lock/burn of protocol fees across all pools
- Potentially permanent reduction of LP fee share if protocol fee was enabled and cannot be turned off

Recommendation:

Add validation and safer ownership handover:

- Reject `_owner == address(0)` unless explicit renounce is intended and safe.
- Prefer a two-step transfer (propose/accept) to prevent accidental bricking.
- If renounce is required, consider requiring protocol fees to be disabled and protocol fees to be fully collected (or explicitly swept) before allowing renounce.

INFO

Flash-based donation mechanism can be front-run by temporary in-range liquidity to capture most of the donation

Locations:

`contracts/interfaces/pool/IUniswapV3PoolActions.sol:83-96`

`contracts/UniswapV3Pool.sol:790-834`

Description:

`flash()` is explicitly documented as a way to “donate underlying tokens pro-rata to currently in-range liquidity providers” by calling it with `amount{0,1}=0` and transferring tokens in the callback.

Because donation distribution is based on the **current in-range liquidity** (`_liquidity = liquidity` at the start of `flash()`), an MEV actor can **front-run** an intended donation transaction by: 1) minting a large, highly concentrated in-range position immediately before the donation, 2) letting the donation execute (which increases `feeGrowthGlobal{0,1}X128`), 3) burning/collecting immediately after.

This can capture the majority of the donated tokens with minimal market risk, effectively diverting donations away from long-term LPs.

Evidence:

Interface advertises donation use-case:

```
/// @dev Can be used to donate underlying tokens pro-rata to currently in-range liquidity
providers by calling
/// with 0 amount{0,1} and sending the donation amount(s) from the callback
function flash(address recipient, uint256 amount0, uint256 amount1, bytes calldata data)
external;
```

Implementation distributes based on current in-range liquidity:

```
uint128 _liquidity = liquidity;
...
feeGrowthGlobal0X128 += FullMath.mulDiv(paid0 - fees0, FixedPoint128.Q128, _liquidity);
feeGrowthGlobal1X128 += FullMath.mulDiv(paid1 - fees1, FixedPoint128.Q128, _liquidity);
```

Impact:

- Intended “donations” can be economically **captured by MEV/JIT liquidity**, reducing the effectiveness of donation/bribing schemes.
- Any external incentive system that relies on `feeGrowthGlobal` as a signal can be manipulated by a donor + JIT attacker combination.

Recommendation:

Donors/integrations should not assume donations will accrue to “current long-term LPs”. If targeted distribution is required, use mechanisms that cannot be diluted by temporary liquidity (e.g., explicit recipient lists, merkle distributions, or time-weighted liquidity eligibility).

INFO

Just-in-time (JIT) liquidity via same-block mint/burn can siphon swap fees from passive LPs (MEV value leakage)

Locations:

`contracts/UniswapV3Pool.sol:301-453`

`contracts/UniswapV3Pool.sol:455-543`

`contracts/UniswapV3Pool.sol:688-691`

`contracts/UniswapV3Pool.sol:757-765`

Description:

The pool's fee model accrues swap fees to *in-range liquidity at the time of the swap* via `feeGrowthGlobal{0,1}X128`. Because liquidity can be added and removed permissionlessly and immediately (mint/burn) and fees are realized when a position is updated, an MEV searcher can:

1) **Mint** a very narrow, high-liquidity position just before a large swap, 2) Let the victim swap execute and pay fees, 3) **Burn** immediately after to realize almost all earned fees, 4) Optionally **collect** and withdraw.

This is the well-known **JIT liquidity** / **liquidity sniping** pattern. It is an economic attack on passive LPs: fees from organic flow are captured by MEV bots that take near-zero price exposure by only being in-range for a single swap/block.

Evidence:

Fees accrue to global fee growth during swap:

```
// contracts/UniswapV3Pool.sol
if (state.liquidity > 0)
 state.feeGrowthGlobalX128 += FullMath.mulDiv(step.feeAmount, FixedPoint128.Q128, state.-
liquidity);
```

(`UniswapV3Pool.sol:688-691`)

Positions realize fees when updated:

```
// contracts/UniswapV3Pool.sol
(uint256 feeGrowthInside0X128, uint256 feeGrowthInside1X128) =
 ticks.getFeeGrowthInside(...);

position.update(liquidityDelta, feeGrowthInside0X128, feeGrowthInside1X128);
```

(`UniswapV3Pool.sol:439-442`)

Liquidity can be added/removed immediately:

```
function mint(...) external override lock returns (uint256 amount0, uint256 amount1) { ... }
function burn(...) external override lock returns (uint256 amount0, uint256 amount1) { ... }
```

(`UniswapV3Pool.sol:455-543`)

Impact:

- **Value leakage over time for passive LPs:** Fee revenue is systematically siphoned by MEV bots around large swaps, reducing returns for LPs providing continuous liquidity.
- **Increased MEV and worse user outcomes:** Searchers compete to win these bundles, potentially increasing effective transaction costs and adverse selection.

Recommendation:

- If the intended design is to reduce JIT liquidity, consider protocol-level mitigations (e.g., fee rebates that depend on time-in-range, cooldowns, or other anti-MEV mechanisms). These are non-trivial and may have tradeoffs.
- At minimum, document JIT liquidity risk for LPs and integrators; encourage LP strategies that account for MEV/toxic flow (e.g., wider ranges, dynamic rebalancing, or using higher fee tiers where appropriate).

INFO

Permissionless pool initialization enables MEV price-setting against first liquidity providers

Location:

`contracts/UniswapV3Pool.sol:269-289`

Description:

`UniswapV3Pool.initialize()` is permissionless and can be called by **any** address exactly once to set the pool's initial `sqrtPriceX96`/tick.

Because `mint()` prices required deposits off of the **current pool price**, an attacker can front-run a user's first liquidity provision by initializing the pool at a deliberately wrong price, causing the victim's subsequent `mint()` (especially if done without tight slippage bounds / atomic initialize+mint) to deposit at an unfavorable ratio.

This is an economic/MEV vector: the attacker can then arbitrage the pool back toward the external "true" price, extracting value from the victim's newly deposited liquidity.

Evidence:

```
// contracts/UniswapV3Pool.sol
function initialize(uint160 sqrtPriceX96) external override {
 require(slot0.sqrtPriceX96 == 0, 'AI');

 int24 tick = TickMath.getTickAtSqrtRatio(sqrtPriceX96);
 ...
 slot0 = Slot0({
 sqrtPriceX96: sqrtPriceX96,
 tick: tick,
 ...
 unlocked: true
 });

 emit Initialize(sqrtPriceX96, tick);
}
```

No access control and no `noDelegateCall`/`lock` modifier is used.

Impact:

- **First LP / first mint economic loss:** A victim who expects to initialize at a fair price (or who mints immediately after someone else initializes) can be forced into an unfavorable initial composition and suffer immediate loss to arbitrage.

- **MEV amplification:** Initializers can observe pending pool deployments and initialization attempts and race to set a price advantageous to their own follow-up trades.

Recommendation:

- Ensure integrations (routers/position managers) perform **create + initialize + mint atomically** where possible, and/or enforce strict **slippage bounds** on minting based on an external price.
- Consider restricting initialization (if the intended design is not fully permission-less) or documenting this risk prominently for integrators/users.

INFO

Pool.observe() / Oracle.observe() has unbounded per-call work and memory allocation based on user-supplied secondsAgos length

Locations:

`contracts/libraries/Oracle.sol:300-324`

`contracts/UniswapV3Pool.sol:235-252`

Description:

`UniswapV3Pool.observe(uint32[] secondsAgos)` forwards the calldata array to `Oracle.observe()`, which: 1) allocates two memory arrays of `secondsAgos.length`, and 2) loops over every element, calling `observeSingle()` each time.

There is **no upper bound** on `secondsAgos.length`. A caller can provide a very large array, causing excessive gas usage and memory expansion, ultimately reverting due to out-of-gas or memory limits.

Although this is an `external view` function (so callers generally bear the cost), it is still callable on-chain from other contracts; any integration that forwards user-controlled `secondsAgos` can be griefed into reverting.

Evidence:

```
// contracts/libraries/Oracle.sol
tickCumulatives = new int56[](secondsAgos.length);
secondsPerLiquidityCumulativeX128s = new uint160[](secondsAgos.length);
for (uint256 i = 0; i < secondsAgos.length; i++) {
 (tickCumulatives[i], secondsPerLiquidityCumulativeX128s[i]) = observeSingle(...);
}
```

```
// contracts/UniswapV3Pool.sol
function observe(uint32[] calldata secondsAgos) external view returns (...) {
 return observations.observe(_blockTimestamp(), secondsAgos, ...);
}
```

Impact:

- **Compute/memory DoS** for `observe()` calls with large arrays (revert).
- **Integration risk:** on-chain callers that do not cap `secondsAgos` length can be forced to fail.

Recommendation:
Consider enforcing a reasonable maximum `secondsAgos` length (or document and encourage callers to cap it) to keep execution within predictable bounds.

INFO

Protocol fee revenue can be reduced via per-step floor division (dust accrues to LPs)

Locations:

`contracts/UniswapV3Pool.sol:681-686`

`contracts/UniswapV3Pool.sol:820-831`

`contracts/UniswapV3Pool.sol:848-865`

Description:

When protocol fees are enabled (`feeProtocol`), the pool computes the protocol's share using **integer floor division** on a **per-swap-step** basis (and similarly in `flash()`), leaving division "dust" to LPs.

Because swaps that traverse many initialized ticks execute many `computeSwapStep()` iterations, the protocol share is computed as:

- `floor(stepFee / feeProtocol)`

instead of (conceptually) `floor(totalFee / feeProtocol)`.

This systematically biases fees **away from the protocol and toward LPs**, and can be (slightly) amplified by concentrating liquidity into many small ranges/ticks so that swaps traverse many steps.

Additionally, `collectProtocol()` deliberately leaves 1 wei behind when withdrawing an entire token's protocol fee balance.

Evidence:

Per-step protocol fee truncation during swaps:

```
// contracts/UniswapV3Pool.sol
if (cache.feeProtocol > 0) {
 uint256 delta = step.feeAmount / cache.feeProtocol; // floor division
 step.feeAmount -= delta;
 state.protocolFee += uint128(delta);
}
```

(around `UniswapV3Pool.sol:681-686`)

Flash protocol fee truncation:

```
uint256 fees0 = feeProtocol0 == 0 ? 0 : paid0 / feeProtocol0; // floor division
...
uint256 fees1 = feeProtocol1 == 0 ? 0 : paid1 / feeProtocol1; // floor division
```

(around `UniswapV3Pool.sol:820-831`)

Permanent 1-wei residue in collectProtocol:

```
if (amount0 == protocolFees.token0) amount0--; // ensure that the slot is not cleared
...
if (amount1 == protocolFees.token1) amount1--; // ensure that the slot is not cleared
```

(around `UniswapV3Pool.sol:856-864`)

Impact:

- **Protocol revenue leakage (small):** Rounding dust from each step is retained by LPs rather than accruing to `protocolFees`. The gap can grow with the number of swap steps (tick crossings).
- **Protocol cannot withdraw the last unit:** `collectProtocol()` keeps a 1-wei residue per token forever (unless more fees accrue), slightly reducing protocol withdrawals.

Recommendation:

- If the intended design is “protocol gets exactly 1/x of total fees”, consider computing protocol fees on the aggregated swap fee amount rather than per-step, or carrying dust forward.
- If the 1-wei residue is not intended, remove the slot-preservation decrement (or replace with a different gas optimization).

INFO

Rounding can yield unchanged sqrt price for nonzero input, leading to 100% fee (no swap) on tiny trades

Locations:

`contracts/libraries/SqrtPriceMath.sol:68-96`

`contracts/libraries/SqrtPriceMath.sol:99-120`

`contracts/libraries/SwapMath.sol:40-97`

Description:

`SqrtPriceMath.getNextSqrtPriceFromAmount1RoundingDown()` (and, in some cases, `getNextSqrtPriceFromAmount0RoundingUp()`) can return the *same* `sqrtPX96` even when `amount > 0`, due to integer truncation/rounding at Q96 precision.

When this happens inside `SwapMath.computeSwapStep()` (exact-input path), `sqrtRatioNextX96` may equal `sqrtRatioCurrentX96` while still being different from `sqrtRatioTargetX96`. The subsequent recomputation of `amountIn` using the delta between `sqrtRatioCurrentX96` and `sqrtRatioNextX96` returns `0`, and the code then charges:

- `feeAmount = uint256(amountRemaining) - amountIn` ' effectively**100% of the user's remaining input** as fees,
- with **0 output**.

This behavior is surprising because it deviates sharply from the nominal `feePips` for small-but-nonzero swaps and can create a "black-hole" path where input is fully consumed without an exchange.

Evidence:

In `SqrtPriceMath.getNextSqrtPriceFromAmount1RoundingDown` (add=true), the quotient is floored:

```
uint256 quotient = (amount <= type(uint160).max)
 ? (amount << FixedPoint96.RESOLUTION) / liquidity
 : FullMath.mulDiv(amount, FixedPoint96.Q96, liquidity);

return uint256(sqrtPX96).add(quotient).toUint160();
```

If `amount * Q96 < liquidity`, then `quotient == 0` and the returned price equals the input price.

Then in `SwapMath.computeSwapStep` (exactIn branch):

```
sqrtRatioNextX96 = SqrtPriceMath.getNextSqrtPriceFromInput(...);
...
amountIn = ... getAmount{0,1}Delta(... sqrtRatioNextX96, sqrtRatioCurrentX96, ...);
...
if (exactIn && sqrtRatioNextX96 != sqrtRatioTargetX96) {
 feeAmount = uint256(amountRemaining) - amountIn;
}
```

If `sqrtRatioNextX96 == sqrtRatioCurrentX96` and `sqrtRatioTargetX96 != sqrtRatioCurrentX96`, then `amountIn == 0` and `feeAmount == amountRemaining`.

Impact:

- Users performing very small exact-input swaps can lose their entire specified input amount and receive 0 output.
- The effective fee can become 100%, far exceeding the configured fee tier.
- While this may be limited to extreme precision/liquidity regimes, the failure mode is an **unexpected one-way asset flow** (input fully consumed, no output) and should be explicitly guarded or documented.

Recommendation:

- Ensure `getNextSqrtPriceFromInput` cannot return `sqrtPX96` when `amountIn > 0` (e.g., enforce a minimum 1-ULP movement), **or**
- In `SwapMath.computeSwapStep`, add a guard: if `sqrtRatioNextX96 == sqrtRatioCurrentX96`, treat as no-op swap and compute fees according to `feePips` (or revert with a clear error), instead of charging the entire remainder as fees.
- At minimum, document a minimum effective swap size / precision limitation so integrators can protect users.

INFO

Swaps can move price/tick (and thus oracle state) with zero in-range liquidity, potentially enabling free oracle manipulation in empty pools

Locations:

`contracts/libraries/SwapMath.sol:21-97`

`contracts/libraries/SqrtPriceMath.sol:145-194`

`contracts/UniswapV3Pool.sol:595-616`

`contracts/UniswapV3Pool.sol:640-752`

Description:

`SwapMath.computeSwapStep` and the pool's swap loop allow the pool price/tick to advance even when `liquidity == 0`, because the computed `amountIn`, `amountOut`, and `feeAmount` become 0 while `sqrtRatioNextX96` is set to the target price.

In `UniswapV3Pool.swap`, there is **no check that in-range liquidity is nonzero**. As a result, an attacker can call `swap()` with any nonzero `amountSpecified` and a permissive `sqrtPriceLimitX96`, and the loop can update `slot0.sqrtPriceX96` and `slot0.tick` **without any token transfers being required** (the callback deltas end up being 0).

Because the oracle accumulators integrate over time using the current tick, this allows an attacker to set the pool's tick/price when liquidity is 0 (e.g., right after `initialize()` and before any mint), then wait for time to pass so that subsequent `observe()` / TWAP consumers see a manipulated price.

Evidence:

Swap step can advance price with zero liquidity:

In `SwapMath.computeSwapStep`, when `liquidity == 0`:

- `SqrtPriceMath.getAmount{0,1}Delta(..., liquidity, ...)` returns 0 because the liquidity multiplier is 0.
- In the exact-in branch:
  - `amountIn` becomes 0
  - `amountRemainingLessFee >= amountIn` holds (since `amountIn == 0`)

- `sqrtRatioNextX96` is set to `sqrtRatioTargetX96`
- `feeAmount` becomes 0

Pool swap does not require liquidity > 0:

`swap()` sets up the loop without checking liquidity:

```
SwapState memory state = SwapState({ ... liquidity: cache.liquidityStart });
while (state.amountSpecifiedRemaining != 0 && state.sqrtPriceX96 != sqrtPriceLimitX96) {
 (state.sqrtPriceX96, step.amountIn, step.amountOut, step.feeAmount) = SwapMath.computeSwap-
Step(
 ...,
 state.liquidity,
 state.amountSpecifiedRemaining,
 fee
 );
 ...
}
```

At the end, if no liquidity existed, the computed swap deltas can remain zero, so the callback payment checks do not enforce any transfer.

Oracle state is updated based on tick/price changes:

After the loop, the pool updates `slot0.tick`/`sqrtPriceX96` and writes an observation if the tick changed:

```
if (state.tick != slot0Start.tick) {
 observations.write(..., slot0Start.tick, cache.liquidityStart, ...);
 slot0.sqrtPriceX96 = state.sqrtPriceX96;
 slot0.tick = state.tick;
} else {
 slot0.sqrtPriceX96 = state.sqrtPriceX96;
}
```

Impact:

- **Oracle manipulation in empty pools**: After `initialize()` but before any mint, or whenever in-range liquidity is 0, an attacker can set the tick/price essentially for free (gas only), then let time elapse to bias TWAP.
- **Downstream protocols** using Uniswap V3 pool oracles without verifying sufficient/liquid liquidity are at risk.

Recommendation:

Consider preventing swaps when `liquidity == 0` (or when no initialized liquidity exists) or otherwise ensure that price/tick updates cannot occur without meaningful economic cost. At minimum, strongly document for oracle consumers that pools with 0/low liquidity are trivially manipulable.

INFO

Tick cumulative interpolation loses precision and can bias TWAP by up to ~1 tick due to division-before-multiplication (round-toward-zero)

Location:

`contracts/libraries/Oracle.sol:271-285`

Description:

When `observeSingle()` is asked for a timestamp that falls **between** two stored observations, it linearly interpolates the accumulators. For `secondsPerLiquidityCumulativeX128` it computes the interpolated delta as `(delta * targetDelta) / observationTimeDelta` (higher precision).

For `tickCumulative`, however, it computes the slope first using integer division and then multiplies:

- `((deltaTickCumulative / observationTimeDelta) * targetDelta)`

Because Solidity signed division rounds **toward zero**, this is **not equivalent** to `(deltaTickCumulative * targetDelta) / observationTimeDelta` and introduces a rounding error that can be as large as almost `targetDelta` tick-seconds. After downstream consumers compute the arithmetic mean tick (divide by `secondsAgo`), this can translate into an average tick that is off by up to ~**1 tick** and is also **biased** (negative deltas round toward zero ' less negative).

This is a business-logic correctness issue for protocols/integrations that depend on precise TWAP boundaries.

Evidence:

```
// Oracle.observeSingle(), "we're in the middle"
return (
 beforeOrAt.tickCumulative +
 ((atOrAfter.tickCumulative - beforeOrAt.tickCumulative) / observationTimeDelta) *
targetDelta,
 beforeOrAt.secondsPerLiquidityCumulativeX128 +
 uint160(
 (uint256(atOrAfter.secondsPerLiquidityCumulativeX128 - beforeOrAt.secondsPerLiquidity-
CumulativeX128)
 * targetDelta) / observationTimeDelta
 )
);
```

Impact:

- TWAP calculations derived from `tickCumulative` can be off by ~1 tick in worst cases (H0.01% price step), especially whe`target` is far from an observation boundary.
- Signed division rounding toward zero creates an asymmetric bias for negative tick cumulatives.
- While small, this can matter in edge-triggered economic logic (liquidation thresholds, TWAP guards, on-chain oracle checks).

Recommendation:

Interpolate `tickCumulative` using full-precision multiply-then-divide (and apply consistent signed rounding rules, e.g., round toward negative infinity) to match the intended linear interpolation semantics.

INFO

Tick stuffing: attacker can initialize many ticks with tiny liquidity to increase swap gas and impede price movement/arbitrage

Locations:

`contracts/libraries/Tick.sol:97-150`

`contracts/UniswapV3Pool.sol:640-651`

Description:

The pool's swap loop steps through **initialized ticks** using the tick bitmap. Any user can mint tiny-liquidity positions that initialize many tick boundaries. This "tick stuffing" increases the number of swap steps (tick crossings / next-initialized-tick searches), raising gas costs and potentially making large price moves impractical within block gas limits.

While this does not directly steal funds, it is an **economic/griefing attack** that can:

- impede arbitrage (keeping the pool price stale/off-market),
- increase traders' costs / cause swap reverts due to out-of-gas,
- create MEV opportunities for the stuffer (trade against a temporarily stale price elsewhere).

Evidence:

Ticks become initialized when liquidityGross goes from 0 to nonzero in `Tick.update()`:

```
flipped = (liquidityGrossAfter == 0) != (liquidityGrossBefore == 0);

if (liquidityGrossBefore == 0) {
 // ... set outside accumulators depending on tick vs current
 info.initialized = true;
}
info.liquidityGross = liquidityGrossAfter;
```

Swaps iterate to the next initialized tick each step:

```
while (state.amountSpecifiedRemaining != 0 && state.sqrtPriceX96 != sqrtPriceLimitX96) {
 (step.tickNext, step.initialized) = tickBitmap.nextInitializedTickWithinOneWord(
 state.tick,
 tickSpacing,
 zeroForOne
 );
 // ... compute step and potentially cross tick
}
```

An attacker can cheaply (relative to the pool TVL) create many narrow/adjacent positions with minimal liquidity to set many bitmap bits.

Impact:

- **Gas DoS / censorship** of swaps that need to traverse many initialized ticks.
- Reduced ability of arbitrageurs to restore the pool price, enabling temporary **price manipulation/staleness** that can be exploited across venues.

Recommendation:

Mitigations are largely design-level and may include:

- operational guidance (use private orderflow for large swaps, monitor tick density),
- parameterization choices (fee tiers/tickSpacing that reduce feasible tick density),
- or additional protocol-level constraints if compatible with Uniswap V3 design goals.

INFO

snapshotCumulativesInside() can be griefed via tick de-initialization (view call reverts when boundary ticks cleared)

Locations:

`contracts/interfaces/pool/IUniswapV3PoolDerivedState.sol:23-39`

`contracts/UniswapV3Pool.sol:157-233`

`contracts/UniswapV3Pool.sol:444-452`

Description:

`snapshotCumulativesInside(tickLower, tickUpper)` hard-reverts unless **both boundary ticks are initialized**.

In the pool implementation, ticks are **cleared** (deleted) when the last liquidity referencing them is removed (`liquidityGross` flips to 0). This means a range that previously existed can become un-queryable via `snapshotCumulativesInside` once all liquidity at either boundary tick is burned.

This creates an economic/griefing vector against *downstream protocols* that rely on being able to take snapshots for a range at arbitrary times (e.g., settlement, reward accounting, governance parameter updates): a user can remove the final unit of liquidity at a boundary tick to make `snapshotCumulativesInside` revert, potentially blocking or delaying those dependent flows.

Evidence:

snapshotCumulativesInside requires initialized ticks:

```
require(initializedLower);
...
require(initializedUpper);
```

Pool clears tick data when liquidity is removed and tick flips to uninitialized:

```
// clear any tick data that is no longer needed
if (liquidityDelta < 0) {
 if (flippedLower) { ticks.clear(tickLower); }
 if (flippedUpper) { ticks.clear(tickUpper); }
}
```

Impact:

- View calls reverting can cause **DoS/griefing** for on-chain systems that assume snapshots are always available for historical ranges.
- If such a system gates value transfers or state transitions on these snapshots, it can become economically exploitable (forced delays, missed reward epochs, etc.).

Recommendation:

Downstream integrators should treat `snapshotCumulativesInside` as only valid while the boundary ticks remain initialized, and design accounting to avoid reliance on it after full liquidity withdrawal (or re-initialize required ticks before snapshotting).

INFO

swap() NatSpec implies exact-output swaps when amountSpecified < 0, but pool may return partial fill without reverting

Locations:

`contracts/interfaces/pool/IUniswapV3PoolActions.sol:65-81`

`contracts/UniswapV3Pool.sol:640-770`

Description:

`IUniswapV3PoolActions.swap()` documents `amountSpecified` as configuring the swap as **exact input** (positive) or **exact output** (negative). In the pool implementation, the swap loop terminates when either the specified amount has been fully consumed **or** the `sqrtPriceLimitX96` is reached. If the price limit is reached before the specified amount is fully satisfied, the function **does not revert**; it returns the amounts actually swapped.

As a result, `amountSpecified < 0` does not guarantee the pool will output exactly `abs(amountSpecified)`; it can produce a smaller output (partial fill) and return early. Likewise, `amountSpecified > 0` may not be fully spent.

Evidence:

Interface NatSpec:

```
/// @param amountSpecified The amount of the swap, which implicitly configures the swap as
exact input (positive), or exact output (negative)
function swap(..., int256 amountSpecified, uint160 sqrtPriceLimitX96, ...) external returns
(int256 amount0, int256 amount1);
```

Implementation terminates on price limit without requiring full satisfaction:

```
while (state.amountSpecifiedRemaining != 0 && state.sqrtPriceX96 != sqrtPriceLimitX96) {
 ...
}

(amount0, amount1) = ... (amountSpecified - state.amountSpecifiedRemaining, state.amountCal-
culated) ...;
```

No `require(state.amountSpecifiedRemaining == 0)` exists.

Impact:

Integrators calling pools directly (without the periphery router's additional checks) may incorrectly assume:

- exact-output swaps always deliver the requested output; or
- exact-input swaps always spend the full specified input.

This can break downstream logic (e.g., settlement, accounting, or slippage protection) if the caller does not validate returned deltas.

Recommendation:

Clarify in interface docs that `amountSpecified` is the target exact input/output **subject to `sqrtPriceLimitX96` and available liquidity**, and callers must verify returned amounts (or revert themselves) when exactness is required.

INFO

Burn/Collect event documentation misrepresents what assets were actually withdrawn vs merely accounted

Locations:

`contracts/interfaces/pool/IUniswapV3PoolEvents.sol:47-62`

`contracts/interfaces/pool/IUniswapV3PoolEvents.sol:31-45`

`contracts/UniswapV3Pool.sol:490-513`

`contracts/UniswapV3Pool.sol:517-543`

Description:

`IUniswapV3PoolEvents` NatSpec for `Burn` and `Collect` can mislead asset accounting:

- `Burn` event docs describe `amount0/amount1` as "withdrawn", but `burn()` **does not transfer any tokens**—it only credits `position.tokensOwed{0,1}`. The actual token transfer happens later via `collect()`.
- `Collect` event docs describe `amount0/amount1` as "fees collected", but `collect()` withdraws **tokens owed** which can come from **both swap fees and burned liquidity (principal)** (as noted in the `collect()` function docs). Labeling all collected amounts as "fees" can cause indexers/accounting systems to misclassify principal withdrawals as fees.

Evidence:

Burn event docs (claims withdrawal):

`contracts/interfaces/pool/IUniswapV3PoolEvents.sol:47-55`

```
/// @notice Emitted when a position's liquidity is removed
/// @param amount0 The amount of token0 withdrawn
/// @param amount1 The amount of token1 withdrawn
event Burn(..., uint256 amount0, uint256 amount1);
```

But burn() only updates tokens owed, no transfers:

`contracts/UniswapV3Pool.sol:535-540`

```
(position.tokensOwed0, position.tokensOwed1) = (
 position.tokensOwed0 + uint128(amount0),
 position.tokensOwed1 + uint128(amount1)
);
```

Collect event docs (labels as fees):

`contracts/interfaces/pool/IUniswapV3PoolEvents.sol:31-45`

```
/// @notice Emitted when fees are collected by the owner of a position
/// @param amount0 The amount of token0 fees collected
/// @param amount1 The amount of token1 fees collected
event Collect(..., uint128 amount0, uint128 amount1);
```

But collect() transfers `tokensOwed{0,1}` (fees + burned liquidity):

`contracts/UniswapV3Pool.sol:500-510`

```
amount0 = ... position.tokensOwed0 ...;
amount1 = ... position.tokensOwed1 ...;
TransferHelper.safeTransfer(token0, recipient, amount0);
TransferHelper.safeTransfer(token1, recipient, amount1);
```

Impact:

- Off-chain accounting/indexers may:
  - treat `Burn.amount{0,1}` as actual outflows when they are not,
  - misclassify principal withdrawals (from burns) as "fees" based on `Collect` event docs.

Recommendation:

Update event NatSpec to reflect actual asset flows:

- `Burn.amount{0,1}`: amounts accounted/owed due to burned liquidity (not transferred).
- `Collect.amount{0,1}`: amounts withdrawn from `tokensOwed{0,1}` (may include fees and burned liquidity).

INFO

Collect event NatSpec omits recipient parameter documentation

Location:

`contracts/interfaces/pool/IUniswapV3PoolEvents.sol:31-45`

Description:

`IUniswapV3PoolEvents.Collect` includes a `recipient` parameter but the NatSpec block does not document it, which can confuse downstream event consumers (indexers, accounting/oracle pipelines).

Evidence:

```
/// @notice Emitted when fees are collected by the owner of a position
/// @param owner The owner of the position for which fees are collected
/// @param tickLower The lower tick of the position
/// @param tickUpper The upper tick of the position
/// @param amount0 The amount of token0 fees collected
/// @param amount1 The amount of token1 fees collected
event Collect(
 address indexed owner,
 address recipient,
 int24 indexed tickLower,
 int24 indexed tickUpper,
 uint128 amount0,
 uint128 amount1
);
```

Impact:

Documentation gaps increase the chance of incorrect event decoding/attribution of fee recipients.

Recommendation:

Add an `@param recipient` NatSpec entry describing the fee recipient address.

INFO

CollectProtocol event NatSpec mislabels token1 amount as `amount0`

Location:

`contracts/interfaces/pool/IUniswapV3PoolEvents.sol:115-121`

Description:

The NatSpec for `CollectProtocol` incorrectly labels the token1 withdrawal amount as `amount0`.

Evidence:

```
// contracts/interfaces/pool/IUniswapV3PoolEvents.sol
/// @param amount0 The amount of token0 protocol fees that is withdrawn
/// @param amount0 The amount of token1 protocol fees that is withdrawn
event CollectProtocol(address indexed sender, address indexed recipient, uint128 amount0,
uint128 amount1);
```

Impact:

Low direct impact, but can cause incorrect decoding/mapping in autogenerated docs, SDKs, or analytics pipelines that use NatSpec tags for field naming, potentially leading to swapped/misreported protocol fee amounts.

Recommendation:

Fix the second tag to `@param amount1` and ensure the description matches token1.

INFO

Factory tickSpacing cap is inconsistent with comment (off-by-one ambiguity)

Location:

`contracts/UniswapV3Factory.sol:61-71`

Description:

`enableFeeAmount()` comments say tick spacing is “capped at 16384”, but the actual check is `tickSpacing < 16384`, which disallows exactly `16384`.

This is a numerical boundary inconsistency: either the code is slightly more restrictive than intended (if `16384` should be allowed), or the comment is misleading.

Evidence:

```
// tick spacing is capped at 16384 ...
require(tickSpacing > 0 && tickSpacing < 16384);
```

Impact:

If `16384` is intended to be valid, it cannot be enabled via the factory, reducing configurability and potentially breaking integrator assumptions based on the comment.

Recommendation:

Align comment and code. Either:

- change the comment to clarify the bound is **strictly less than** `16384`, or
- (if safe) allow `tickSpacing == 16384` by using `<= 16384`.

INFO

Flash event `paid0/paid1` represent net balance increase (fee + extra), not total repayment of principal + fee (NatSpec is ambiguous/misleading)

Locations:

`contracts/interfaces/pool/IUniswapV3PoolEvents.sol:82-96`

`contracts/UniswapV3Pool.sol:800-819`

Description:

`IUniswapV3PoolEvents.Flash` documents `paid0/paid1` as the amounts "paid for the flash". Many integrators will naturally interpret this as **the total amount repaid** (principal + fee). In the implementation, `paid0/paid1` are computed as `balanceAfter - balanceBefore` where `balanceBefore` is measured **before** the flash transfer is sent out.

That means `paid0/paid1` are the pool's **net gain** (fee plus any extra donation), and typically **do not include the flashed principal**.

The NatSpec line "can exceed the amount0 plus the fee" is particularly misleading under the net-gain interpretation, because the common/expected case is `paid0 ≈ fee0` (much smaller than `amount0`).

Evidence:

Interface (`contracts/interfaces/pool/IUniswapV3PoolEvents.sol`):

```
/// @param amount0 The amount of token0 that was flashed
/// @param paid0 The amount of token0 paid for the flash, which can exceed the amount0 plus the fee
```

Implementation (`contracts/UniswapV3Pool.sol`):

```
uint256 fee0 = FullMath.mulDivRoundingUp(amount0, fee, 1e6);
uint256 balance0Before = balance0();
...
if (amount0 > 0) TransferHelper.safeTransfer(token0, recipient, amount0);
...
uint256 balance0After = balance0();
require(balance0Before.add(fee0) <= balance0After, 'F0');
uint256 paid0 = balance0After - balance0Before;
emit Flash(msg.sender, recipient, amount0, amount1, paid0, paid1);
```

Here, `paid0` is `balanceAfter - balanceBefore`, i.e., fee + any extra amount returned above the original balance, not principal+fee.

Impact:

Off-chain analytics/indexers and direct integrators may miscompute flash repayments/fees if they assume `paid0` includes principal.

Recommendation:

Clarify NatSpec that `paid0/paid1` are the **net amounts paid into the pool** (fee + any extra), i.e., `balanceAfter - balanceBefore`, and typically equal the fee unless additional tokens are sent in.

INFO

IERC20Minimal/interface documentation claims strict ERC20 compliance, but core transfer helper explicitly supports non-standard return values (integration assumption mismatch)

Locations:

`contracts/interfaces/IERC20Minimal.sol:12-39`

`contracts/interfaces/IUniswapV3Pool.sol:12-14`

`contracts/libraries/TransferHelper.sol:6-22`

Description:

Several interface comments/documentation state that pool assets must "strictly conform to the ERC20 specification" and that `transfer/transferFrom/approve` return `bool`:

- `IERC20Minimal.transfer(...) external returns (bool);`
- `IUniswapV3Pool` comment: "assets that strictly conform to the ERC20 specification"

However, `TransferHelper.safeTransfer` explicitly accepts tokens that return **no value** (empty returndata):

```
require(success && (data.length == 0 || abi.decode(data, (bool))), 'TF');
```

This means the implementation is intentionally looser than the interface documentation suggests.

Impact:

This mismatch can lead integrators/users to:

- Assume only strict ERC20s are supported, when some legacy/non-standard tokens may work.
- More importantly, misunderstand the *actual* safety boundary: returning no data is treated as success, which increases the risk surface for broken tokens (see separate finding on silent success without transfer).

While primarily a documentation issue, misunderstanding token assumptions can cause **stuck funds** or unexpected behavior when interacting with pools involving non-standard tokens.

Recommendation:

Align docs with implementation by clarifying:

- Whether non-standard ERC20 return values are supported.
- Which token behaviors are explicitly unsupported (fee-on-transfer, rebasing, malicious balanceOf/transfer semantics), and the associated asset-loss/stuck-fund risks.

INFO

LICENSE Change Date (2023-04-01) appears elapsed while source files still declare BUSL-1.1 SPDX

Locations:

`contracts/libraries/LICENSE:363-381`

`contracts/libraries/Oracle.sol:1`

Description:

`contracts/libraries/LICENSE` specifies a **Change Date** of "the earlier of 2023-04-01 or a date specified at v3-core-license-date.uniswap.eth" and states that **effective on the Change Date** the work is granted under the **Change License** (GPL v2.0 or later).

Given the audit date (2026-02-22), the hard-coded Change Date shown in the repository text is in the past. However, `Oracle.sol` (and other files) still use the SPDX identifier `BUSL-1.1`, which can mislead downstream users and compliance tooling about the effective license.

This is distinct from the earlier "dual license choice" ambiguity: this issue is specifically about the **Change Date having passed** vs the **SPDX headers not reflecting the change**.

Evidence:

From LICENSE:

```
Change Date:          The earlier of 2023-04-01 or a date specified at
 v3-core-license-date.uniswap.eth
Change License:       GNU General Public License v2.0 or later
...
Effective on the Change Date ... the Licensor hereby grants you rights under
the terms of the Change License...
```

(contracts/libraries/LICENSE:363-381)

From Oracle.sol:

```
// SPDX-License-Identifier: BUSL-1.1
```

(contracts/libraries/Oracle.sol:1)

Impact:

Compliance ambiguity: automated scanners and integrators may treat the code as BUSL-restricted when the bundled LICENSE text indicates it has converted to GPL as of the Change Date.

Recommendation:

Align SPDX identifiers, top-level license documentation, and any ENS-based change-date mechanism so that the effective license is unambiguous for the current version.

INFO

LiquidityMath.addDelta NatSpec return description is incorrect (returns new liquidity, not delta)

Location:

`contracts/libraries/LiquidityMath.sol:6-15`

Description:

`LiquidityMath.addDelta` returns the **new liquidity value after applying the delta**, but the NatSpec says it returns “The liquidity delta”.

This is a semantic documentation bug: consumers reading the library may assume the return is the signed delta or the change amount rather than the updated liquidity.

Evidence:

```
/// @return z The liquidity delta
function addDelta(uint128 x, int128 y) internal pure returns (uint128 z) {
 if (y < 0) {
 require((z = x - uint128(-y)) < x, 'LS');
 } else {
 require((z = x + uint128(y)) >= x, 'LA');
 }
}
```

Impact:

Misleading documentation increases the risk of integrator mistakes and incorrect downstream usage (e.g., treating the return as a delta).

Recommendation:

Update the NatSpec to state that `z` is the resulting liquidity after applying `y` to `x`.

INFO

Mint/Swap callbacks accept overpayment with no refund path (accidental donation can become unrecoverable)

Locations:

`contracts/interfaces/pool/IUniswapV3PoolActions.sol:12-29`

`contracts/interfaces/pool/IUniswapV3PoolActions.sol:65-81`

`contracts/UniswapV3Pool.sol:478-485`

`contracts/UniswapV3Pool.sol:771-784`

Description:

For `mint()` and `swap()`, the pool only checks that the post-callback balance increased by **at least** the required amount. Any excess token transfers made by the callback are accepted and retained by the pool (effectively a donation).

The interfaces describe paying “any token owed”, but do not clearly warn that **overpayment is not refunded**. This is an asset-management footgun for integrators implementing callbacks that may accidentally transfer too much (rounding, approvals, or fee-on-transfer quirks), causing unrecoverable loss for the payer.

Evidence:

Mint balance check allows overpay:

`contracts/UniswapV3Pool.sol:478-485`

```
if (amount0 > 0) balance0Before = balance0();
if (amount1 > 0) balance1Before = balance1();
IUniswapV3MintCallback(msg.sender).uniswapV3MintCallback(amount0, amount1, data);
if (amount0 > 0) require(balance0Before.add(amount0) <= balance0(), 'M0');
if (amount1 > 0) require(balance1Before.add(amount1) <= balance1(), 'M1');
```

Swap balance check allows overpay:

`contracts/UniswapV3Pool.sol:771-784`

```
uint256 balance0Before = balance0();
IUniswapV3SwapCallback(msg.sender).uniswapV3SwapCallback(amount0, amount1, data);
require(balance0Before.add(uint256(amount0)) <= balance0(), 'IIA');
```

(and similarly for token1)

Impact:

- Callback implementers can accidentally “donate” funds to the pool with no direct recovery path.
- With non-standard tokens (fee-on-transfer, rebasing), naïve callback code that assumes `transferFrom(amount)` results in `amount` received can overpay/underpay unpredictably.

Recommendation:

Explicitly document in the interfaces/spec that:

- The pool verifies minimum payment only.
- Any overpayment is retained by the pool (donation) and is not refunded.

INFO

NatSpec in observe() describes returning 'liquidity' though output is secondsPerLiquidity cumulative

Location:

`contracts/interfaces/pool/IUniswapV3PoolDerivedState.sol:8-21`

Description:

`IUniswapV3PoolDerivedState.observe()` NatSpec says it returns “cumulative tick and liquidity”, but the second returned array is **cumulative seconds-per-liquidity-in-range** (`secondsPerLiquidityCumulativeX128s`), not liquidity.

While related (liquidity-in-range can be derived from differences), the wording is inaccurate and can lead oracle consumers to misuse the output.

Evidence:

```
/// @notice Returns the cumulative tick and liquidity as of each timestamp `secondsAgo`...
/// @return secondsPerLiquidityCumulativeX128s Cumulative seconds per liquidity-in-range value
...
function observe(uint32[] calldata secondsAgos)
 external
 view
 returns (int56[] memory tickCumulatives, uint160[] memory secondsPerLiquidityCumula-
tiveX128s);
```

Impact:

Off-chain and on-chain consumers may treat the second return value as cumulative liquidity rather than cumulative seconds/liquidity, resulting in incorrect TWAP/liquidity calculations.

Recommendation:

Update the summary NatSpec (`@notice`) to explicitly state the function returns **cumulative tick** and **cumulative seconds per liquidity-in-range**.

INFO

NatSpec return documentation for snapshotCumulativesInside() mis-describes secondsInside

Location:

`contracts/interfaces/pool/IUniswapV3PoolDerivedState.sol:23-39`

Description:

`IUniswapV3PoolDerivedState.snapshotCumulativesInside()` claims `secondsInside` is "The snapshot of seconds per liquidity for the range", but the implementation returns **seconds spent inside the tick range** (not seconds-per-liquidity).

This is an oracle/external-data footgun: off-chain indexers and integrators can misinterpret the returned value and compute incorrect time-in-range metrics.

Evidence:

Interface NatSpec:

```
/// @return secondsInside The snapshot of seconds per liquidity for the range
function snapshotCumulativesInside(int24 tickLower, int24 tickUpper)
 external
 view
 returns (
 int56 tickCumulativeInside,
 uint160 secondsPerLiquidityInsideX128,
 uint32 secondsInside
 );
```

Implementation returns `time - secondsOutsideLower - secondsOutsideUpper` (seconds in-range):

```
// contracts/UniswapV3Pool.sol
return (
 tickCumulative - tickCumulativeLower - tickCumulativeUpper,
 secondsPerLiquidityCumulativeX128 - secondsPerLiquidityOutsideLowerX128 - secondsPerLiq-
uidityOutsideUpperX128,
 time - secondsOutsideLower - secondsOutsideUpper
);
```

Impact:

Misleading interface documentation can cause consumers to treat `secondsInside` as scaled by liquidity, producing incorrect accounting/analytics and potentially incorrect downstream oracle computations.

Recommendation:
Fix the NatSpec for `secondsInside` to clearly state it is the **seconds spent inside the tick range** (unscaled), and (optionally) link to how it is computed in the pool/Oracle logic.

INFO

Oracle cumulative outputs are modulo-limited (uint32/uint160/int56 wrap) but interfaces do not warn consumers about overflow/wrap-around arithmetic

Locations:

`contracts/interfaces/pool/IUniswapV3PoolDerivedState.sol:8-22`

`contracts/interfaces/pool/IUniswapV3PoolState.sol:99-115`

`contracts/UniswapV3Pool.sol:132-135`

`contracts/libraries/Oracle.sol:23-45`

Description:

The pool's oracle and derived-state numerics intentionally use fixed-width accumulators:

- timestamps are truncated to `uint32` (`block.timestamp mod 2^32`),
- `secondsPerLiquidityCumulativeX128` is `uint160`, and
- `tickCumulative` is `int56`.

These values can **overflow/wrap-around** (modulo their bit width) over long time horizons and/or low liquidity. The implementation explicitly relies on this behavior (e.g., `_blockTimestamp()` truncation and arithmetic in `Oracle.transform`).

The interfaces (`IUniswapV3PoolDerivedState.observe` and `IUniswapV3PoolState.observations`) do not warn consumers that these outputs are modulo-limited and can wrap, which can cause **incorrect numerical results** for naive off-chain/on-chain consumers that don't use modulo-difference logic.

Evidence:

Timestamp truncation:

```
// contracts/UniswapV3Pool.sol
function _blockTimestamp() internal view virtual returns (uint32) {
 return uint32(block.timestamp); // truncation is desired
}
```

Accumulator updates (wrap is implicit in Solidity 0.7 due to casts and fixed-width types):

```
// contracts/libraries/Oracle.sol
uint32 delta = blockTimestamp - last.blockTimestamp;
...
tickCumulative: last.tickCumulative + int56(tick) * delta,
secondsPerLiquidityCumulativeX128: last.secondsPerLiquidityCumulativeX128 + ((uint160(delta)
<< 128) / (liquidity > 0 ? liquidity : 1)),
```

Interface descriptions present these as regular “as of timestamp” cumulative values without mentioning modulo behavior:

```
// contracts/interfaces/pool/IUniswapV3PoolDerivedState.sol
/// @notice Returns the cumulative tick and liquidity as of each timestamp `secondsAgo` from
the current block timestamp
```

Impact:

Consumers computing TWAPs and other time-weighted metrics may produce incorrect results around wrap-around boundaries (notably ~2^32 seconds H 136 years for timestamps, and potentially for `uint160` seconds-per-liquidity at very low liquidity), if they assume monotonic, non-wrapping accumulators.

Recommendation:

Document that oracle timestamps are `uint32`-truncated and that the cumulative outputs are modulo-limited and may wrap, and recommend using modulo-safe difference calculations when comparing cumulative values.

INFO

Oracle observation math only safe for <=1 uint32 timestamp overflow (oracle may break after long uptime)

Locations:

`contracts/libraries/Oracle.sol:23-45`

`contracts/libraries/Oracle.sol:122-140`

`contracts/libraries/Oracle.sol:245-286`

Description:

`Oracle` truncates timestamps to `uint32` and relies on wraparound arithmetic. Multiple functions explicitly assume timestamps are only subject to **0 or 1 overflows** (i.e., the pool is queried/operated within a window smaller than `2**32` seconds from the relevant reference).

After more than one wrap of `uint32` timestamps (or any scenario violating the “0 or 1 overflow” assumption), observation ordering/comparisons can become incorrect, which can lead to wrong accumulator values returned by `observe()/observeSingle()` and potentially out-of-gas/infinite-loop behavior in `binarySearch()` due to broken invariants.

Even with a single overflow, accumulator types (`int56`, `uint160`) are intentionally narrow and depend on modulo arithmetic; this is safe only as long as consumers’ time windows remain within the intended bounds.

Evidence:

The library documents and depends on the limitation:

```
// Oracle.sol
// @dev blockTimestamp _must_ be chronologically equal to or greater than last.blockTimestamp,
safe for 0 or 1 overflows
uint32 delta = blockTimestamp - last.blockTimestamp;
```

(Oracle.sol:23-37)

```
// @dev safe for 0 or 1 overflows, a and b _must_ be chronologically before or equal to time
function lte(uint32 time, uint32 a, uint32 b) private pure returns (bool) { ... }
```

(Oracle.sol:122-140)

`observeSingle()` and `binarySearch()` depend on correct chronological comparisons derived from `lte()`:

```
(Observation memory beforeOrAt, Observation memory atOrAfter) =
 getSurroundingObservations(self, time, target, tick, index, liquidity, cardinality);
...
uint32 observationTimeDelta = atOrAfter.blockTimestamp - beforeOrAt.blockTimestamp;
```

(Oracle.sol:245-286)

Impact:

- **Incorrect oracle outputs** (tick cumulative / seconds-per-liquidity cumulative) once the timestamp wrap assumptions are violated.
- Protocol components relying on these values (e.g., tick-cross accounting and derived TWAP consumers) may compute incorrect results.
- In the worst case (broken ordering assumptions), `binarySearch()`'s unbounded `while (true)` loop may not converge, causing OOG reverts for `observe()` calls.

While the trigger requires extreme uptime or violating documented preconditions, the potential impact is broad if it ever occurs.

Recommendation:

- If long-lived deployments are a requirement, remove or mitigate the 0/1-overflow assumption (e.g., store larger timestamps or track epoch/overflow count).
- At minimum, make the limitation explicit in user-facing docs and consider adding defensive checks to avoid non-terminating search behavior if invariants are broken.

INFO

Oracle secondsPerLiquidityCumulativeX128 can overflow uint160 at extreme long-lived/low-liquidity boundary

Locations:

`contracts/libraries/Oracle.sol:30-45`

`contracts/libraries/Oracle.sol:279-285`

Description:

`Oracle.Observation.secondsPerLiquidityCumulativeX128` is stored as `uint160` and incremented by `((delta << 128) / max(1, liquidity))`.

At extreme boundaries (very long-lived pools and very small `liquidity`), this cumulative can reach `2^160` and **overflow/wrap** (Solidity 0.7.x has unchecked arithmetic for `uint160`). The rest of the oracle code assumes these accumulators are monotonically increasing and uses plain subtraction to compute deltas; if the accumulator wraps, delta computations underflow/wrap and return incorrect values.

Evidence:

```
// Oracle.transform
uint32 delta = blockTimestamp - last.blockTimestamp;
secondsPerLiquidityCumulativeX128: last.secondsPerLiquidityCumulativeX128 +
 ((uint160(delta) << 128) / (liquidity > 0 ? liquidity : 1)),
```

and later in interpolation:

```
atOrAfter.secondsPerLiquidityCumulativeX128 - beforeOrAt.secondsPerLiquidityCumulativeX128
```

Impact:

For pools that persist across extremely long timescales (on the order of 2^32 seconds H 136 years) with low in-range liquidity (worst cas`liquidity == 0` uses denominator 1), `secondsPerLiquidityCumulativeX128` can overflow `uint160`, producing incorrect oracle observations and derived values (e.g., `observe()` / `snapshotCumulativesInside()` results).

Recommendation:
If long-lived correctness is a goal, either:

- widen the accumulator type (e.g., `uint256`), or
- explicitly document that the oracle is only safe before accumulator wrap, or
- implement modular-difference handling similar to the 32-bit timestamp overflow logic.

INFO

Oracle.binarySearch can run indefinitely / underflow if caller violates timestamp bounds assumptions

Locations:

`contracts/libraries/Oracle.sol:153-184`

`contracts/libraries/Oracle.sol:122-140`

Description:

`Oracle.binarySearch()` uses `while (true)` and relies on several unstated-but-critical preconditions:

- `target` must be within the chronological bounds of stored observations.
- The observation ring must contain at least one initialized observation and the chosen search interval must contain a valid `[beforeOrAt, atOrAfter]` bracket.
- The `lte()` comparator assumptions must hold (inputs chronologically <= `time`, safe for 0/1 overflows).

If a caller violates these assumptions (e.g., passes an out-of-range `target`, inconsistent `index/cardinality`, or observations with unexpected initialization patterns), the loop may not find a bracket and can:

- spin until out-of-gas, or
- hit `r = i - 1` with `i == 0`, underflowing `r` to `2^256-1` and making termination even less likely.

This is an input validation / robustness issue: the function does not defensively enforce its preconditions.

Evidence:

```
function binarySearch(...) private view returns (Observation memory beforeOrAt, Observation
memory atOrAfter) {
 uint256 l = (index + 1) % cardinality;
 uint256 r = l + cardinality - 1;
 uint256 i;
 while (true) {
 i = (l + r) / 2;
 beforeOrAt = self[i % cardinality];
 if (!beforeOrAt.initialized) {
 l = i + 1;
 continue;
 }
 atOrAfter = self[(i + 1) % cardinality];
 bool targetAtOrAfter = lte(time, beforeOrAt.blockTimestamp, target);
 if (targetAtOrAfter && lte(time, target, atOrAfter.blockTimestamp)) break;
 if (!targetAtOrAfter) r = i - 1; // potential underflow if i==0
 else l = i + 1;
 }
}
```

Impact:

For Uniswap V3 core's intended usage, upstream checks in `getSurroundingObservations()` are designed to keep inputs in-range, so this should not occur under normal operation.

However, lack of defensive validation means:

- any misuse (forks, refactors, or external wrapper contracts) can cause oracle reads to become an out-of-gas DoS;
- corrupted observation state (whether via unexpected initialization sequences) can create hard-to-diagnose failures.

Recommendation:

Either:

- add explicit bounds/termination guards (e.g., verify target bounds, enforce monotonicity, enforce at least one initialized observation in interval, and/or use a bounded loop), or
- strengthen documentation that `binarySearch` must only be called after verifying bounds, and assert those bounds in debug builds/tests.

INFO

Oracle.sol documentation misstates meaning of `cardinality` (population vs capacity), risking incorrect integrations

Locations:

`contracts/libraries/Oracle.sol:65-78`

`contracts/UniswapV3Pool.sol:61-67`

Description:

`Oracle.sol` repeatedly documents `cardinality` as “the number of populated elements in the oracle array”, but the UniswapV3Pool uses/persists `observationCardinality` as “the current maximum number of observations that are being stored” (i.e., **capacity**, not necessarily the number of initialized entries).

The implementation in `Oracle` also behaves like `cardinality` is a capacity (ring-buffer modulus, potentially containing uninitialized slots after `grow()`), and it contains special handling for uninitialized slots.

This mismatch can mislead integrators reusing `Oracle` and passing the wrong value as `cardinality`, which can cause incorrect oracle reads, unexpected reverts (e.g., `OLD`), or non-termination/OOG in the search logic if invariants are violated.

Evidence:

Oracle.sol function docs:

```
/// @param cardinality The number of populated elements in the oracle array
function write(..., uint16 cardinality, uint16 cardinalityNext)
```

(contracts/libraries/Oracle.sol:73-78)

Pool’s persisted meaning:

```
// UniswapV3Pool.Slot0
// the current maximum number of observations that are being stored
uint16 observationCardinality;
// the next maximum number of observations to store, triggered in observations.write
uint16 observationCardinalityNext;
```

(contracts/UniswapV3Pool.sol:63-66)

Impact:

Primarily a **developer/integration footgun**: third-party contracts using this library could mis-track cardinality and get incorrect oracle outputs or reverts.

Recommendation:

Update Oracle.sol NatSpec to clearly define:

- `cardinality`: current observation buffer capacity (may include uninitialized slots)
- `cardinalityNext`: requested future capacity

and document how uninitialized slots are handled during the growth phase.

INFO

Oracle.write lacks defensive validation of index/cardinality invariants (can revert or corrupt oracle data if misused)

Location:

`contracts/libraries/Oracle.sol:78-101`

Description:

`Oracle.write()` assumes the caller maintains several critical invariants (documented in comments) but **does not validate them**:

- `cardinalityUpdated` must be > 0 (otherwise `(index + 1) % cardinalityUpdated` is modulo-by-zero).
- `index` should refer to the most-recently-written initialized observation within the logical ring buffer.

If the caller passes inconsistent values (e.g., calling before initialization with `cardinality == 0`, or passing an `index` that points to an uninitialized slot), the function may:

- revert unexpectedly (division/modulo by zero), or
- write an observation derived from an uninitialized `last` observation, producing incorrect cumulative values.

This is an input validation gap inside a library that is relied upon for correctness.

Evidence:

```
function write(
 Observation[65535] storage self,
 uint16 index,
 uint32 blockTimestamp,
 int24 tick,
 uint128 liquidity,
 uint16 cardinality,
 uint16 cardinalityNext
) internal returns (uint16 indexUpdated, uint16 cardinalityUpdated) {
 Observation memory last = self[index];

 if (last.blockTimestamp == blockTimestamp) return (index, cardinality);

 if (cardinalityNext > cardinality && index == (cardinality - 1)) {
 cardinalityUpdated = cardinalityNext;
 } else {
 cardinalityUpdated = cardinality;
 }

 indexUpdated = (index + 1) % cardinalityUpdated; // <--- requires cardinalityUpdated > 0
 self[indexUpdated] = transform(last, blockTimestamp, tick, liquidity); // <--- assumes
`last` is a valid initialized observation
}
```

Impact:

- In any integration that accidentally calls `write()` before initialization or with corrupted/incorrect state variables, oracle updates can revert (DoS for the calling function) or produce incorrect oracle observations (mispricing for TWAP consumers).
- In Uniswap V3 core, these invariants are intended to be maintained by `UniswapV3Pool` state; however, the lack of defense-in-depth means any future refactor, fork, or reuse of this library has a sharper failure mode.

Recommendation:

Add defensive checks (e.g., `require(cardinality > 0)` and/or `require(last.initialized)` / bounds) or ensure callers *always* gate `write()` behind initialization and correct index/cardinality tracking. At minimum, document the invariants more explicitly as *must-hold* preconditions.

INFO

Oracle.write/transform NatSpec suggests passing end-of-interval tick/liquidity, but correct semantics require pre-change values

Locations:

`contracts/libraries/Oracle.sol:23-29`

`contracts/libraries/Oracle.sol:65-76`

`contracts/UniswapV3Pool.sol:732-742`

`contracts/UniswapV3Pool.sol:340-348`

Description:

`Oracle.transform()` and `Oracle.write()` NatSpec describe `tick` and `liquidity` as values “at the time of the new observation”. Semantically, however, the implementation uses these parameters to integrate over the **elapsed time since the previous observation** (`delta = blockTimestamp - last.blockTimestamp`).

That means the correct inputs are the tick/liquidity that were active **during the elapsed interval**, which—when a state change is about to happen in the current transaction—are typically the *pre-change* values.

This is easy to misuse in other integrations because the documentation reads like you should pass the tick/liquidity *at the end* of the interval (i.e., after you update state), which would compute cumulatives using the wrong tick/liquidity.

Evidence:

Oracle side (note `delta` is applied using the passed `tick` and `liquidity`):

```
// contracts/libraries/Oracle.sol
uint32 delta = blockTimestamp - last.blockTimestamp;
return Observation({
 blockTimestamp: blockTimestamp,
 tickCumulative: last.tickCumulative + int56(tick) * delta,
 secondsPerLiquidityCumulativeX128: last.secondsPerLiquidityCumulativeX128 +
 ((uint160(delta) << 128) / (liquidity > 0 ? liquidity : 1)),
 initialized: true
});
```

Pool call sites demonstrate the *actual* required semantics (pass pre-change values):

```
// contracts/UniswapV3Pool.sol (swap)
observations.write(
 slot0Start.observationIndex,
 cache.blockTimestamp,
 slot0Start.tick,         // tick before swap-induced tick change
 cache.liquidityStart,    // liquidity before swap-induced liquidity change
 ...
);
```

(and similarly in `_modifyPosition` before mutating `liquidity`)

Impact:

Documentation/code semantic mismatch can lead to incorrect oracle accumulator values in any downstream protocol that reuses `Oracle.sol` but follows the NatSpec literally (passing post-change tick/liquidity).

Within this repository, `UniswapV3Pool` uses it correctly, so the bug is primarily a correctness footgun for reuse and auditing.

Recommendation:

Clarify NatSpec for `transform()`/`write()` that `tick` and `liquidity` must correspond to the values that were active over the elapsed interval since the last observation (typically the pre-update values when writing in the middle of a state transition).

INFO

Permissionless oracle observation growth allows permanent state bloat and marginal gas-griefing for on-chain consumers

Locations:

`contracts/interfaces/pool/IUniswapV3PoolActions.sol:98-102`

`contracts/UniswapV3Pool.sol:254-267`

`contracts/libraries/Oracle.sol:103-120`

`contracts/libraries/Oracle.sol:153-184`

Description:

`increaseObservationCardinalityNext()` is permissionless and allows anyone to increase the pool's oracle ring-buffer capacity up to `uint16` max.

Although the caller pays the one-time SSTORE loop in `Oracle.grow`, the resulting state size increase is **permanent** and slightly increases the work done by `observe()` (binary search over a larger ring buffer). This can be used for **economic griefing** against protocols that call `observe()` *on-chain* under tight gas constraints, and contributes to state bloat.

Evidence:

Permissionless action:

```
function increaseObservationCardinalityNext(uint16 observationCardinalityNext) external;
```

Pool exposes it without access control:

```
function increaseObservationCardinalityNext(uint16 observationCardinalityNext) external override lock {
 ... observations.grow(observationCardinalityNextOld, observationCardinalityNext);
}
```

Growth loops over every new slot:

```
for (uint16 i = current; i < next; i++) self[i].blockTimestamp = 1;
```

observe() uses binary search across the ring buffer (more observations => more work):

```
function binarySearch(...) private view returns (...) {
 uint256 l = (index + 1) % cardinality;
 uint256 r = l + cardinality - 1;
 while (true) { ... }
}
```

Impact:

- **Permanent state bloat** for pools (larger observation arrays).
- Slightly higher gas for `observe()` calls (particularly relevant if called from other contracts).

Recommendation:

Downstream protocols should avoid tight gas assumptions around `observe()` and consider bounding `secondsAgos` usage. If state bloat is a concern for a specific deployment, consider governance/policy restricting who can call `increaseObservationCardinalityNext()`.

INFO

Pool has no recovery/skim mechanism for accidentally sent tokens or overpayments (assets can be trapped)

Locations:

```
contracts/UniswapV3Pool.sol:137-155
```

```
contracts/UniswapV3Pool.sol:455-868
```

Description:

`UniswapV3Pool` holds ERC-20 assets (`token0`/`token1`) and performs transfers out only as part of swap/mint/flash settlement and fee/protocol-fee collection. There is **no generic withdrawal/rescue/skim** function for:

- ERC-20 tokens mistakenly transferred directly to the pool (including tokens other than `token0`/`token1`)
- excess `token0`/`token1` accidentally overpaid during callbacks (mint/swap/flash require *at least* the owed amount, so overpayment is accepted)
- forced ETH sent via `SELFDESTRUCT` (no `receive()`/`fallback()` exists)

These assets can enter the contract but do not have a corresponding “outlet”.

Evidence:

Outflows are limited to `TransferHelper.safeTransfer` usages in `collect`, `swap`, `flash`, and `collectProtocol`:

- `collect`: transfers `token0`/`token1` based on `position.tokensOwed{0,1}`
- `swap`: transfers output token amounts
- `flash`: transfers principal to `recipient`
- `collectProtocol`: transfers `protocolFees`

There is no function to withdraw arbitrary ERC-20s or ETH.

Impact:

Users or integrators who mistakenly transfer tokens (or overpay during callbacks) can lose access to those assets permanently.

Recommendation:

Consider adding a controlled recovery/skim mechanism (even if only for non-`token0`-/`token1` assets or for forced ETH), or explicitly document the irreversible nature of accidental transfers/overpayments.

INFO

Pool-level mint/burn lack built-in slippage bounds, enabling MEV manipulation of LP entry/exit amounts for naive callers

Locations:

```
contracts/UniswapV3Pool.sol:455-487
contracts/UniswapV3Pool.sol:517-543
contracts/UniswapV3Pool.sol:306-372
```

Description:

At the pool level, `mint()` and `burn()` compute required/returned token amounts purely from the **current pool price** (`slot0.sqrtPriceX96`, `slot0.tick`) and the requested tick range/liquidity delta.

The pool does **not** include any parameters for callers to enforce:

- maximum token amounts for `mint()`, or
- minimum/expected amounts for `burn()`.

As a result, a naive caller who quotes expected deposit/withdraw amounts off-chain and then submits a transaction can be **MEV-sandwiched**:

- An attacker front-runs with a swap to move the pool price,
- The victim's `mint()`/`burn()` executes at the manipulated price, changing the victim's token composition,
- The attacker back-runs to restore price.

While sophisticated integrations can enforce bounds inside the callback (and common periphery contracts do), the pool itself offers no built-in protection, so direct pool interactions or poorly designed wrappers are economically fragile.

Evidence:

`mint()` takes only `(recipient, tickLower, tickUpper, liquidityAmount, data)` and then computes `amount0/amount1` from `_modifyPosition` using current `slot0`:

```
// contracts/UniswapV3Pool.sol
function mint(...) external override lock returns (uint256 amount0, uint256 amount1) {
 ...
 (, int256 amount0Int, int256 amount1Int) = _modifyPosition(...);
 amount0 = uint256(amount0Int);
 amount1 = uint256(amount1Int);
 ...
 IUniswapV3MintCallback(msg.sender).uniswapV3MintCallback(amount0, amount1, data);
 ...
}
```

Amounts depend on `_modifyPosition()` logic:

```
// contracts/UniswapV3Pool.sol
if (_slot0.tick < params.tickLower) { amount0 = ... }
else if (_slot0.tick < params.tickUpper) { amount0 = ...; amount1 = ... }
else { amount1 = ... }
```

(`UniswapV3Pool.sol:327-370`)

`burn()` similarly computes returned amounts from current price via `_modifyPosition()`.

Impact:

- **LP value loss / forced composition change** for naive callers or wrappers that do not enforce slippage bounds.
- **MEV amplification** around large liquidity provision/removal events.

Recommendation:

- Ensure integrations enforce slippage and price bounds at a higher layer (e.g., in the mint/burn caller contract or callback logic).
- Document clearly that pool-level actions are low-level primitives and are not safe for direct use without slippage protection.

INFO

Position key derivation uses abi.encodePacked but interface docs do not specify encoding (can cause wrong key off-chain)

Locations:

`contracts/libraries/Position.sol:30-37`

`contracts/interfaces/pool/IUniswapV3PoolState.sol:81-88`

Description:

The canonical position key is derived with `keccak256(abi.encodePacked(owner, tickLower, tickUpper))`.

However, the `IUniswapV3PoolState.positions(bytes32 key)` NatSpec only states that the key is a hash of a preimage composed of those fields, without specifying **packed** encoding.

Because `abi.encode(...)` vs `abi.encodePacked(...)` produce different byte representations, an integrator who derives `key` using standard ABI encoding may query the wrong storage slot and interpret the position as empty.

Evidence:

```
// contracts/libraries/Position.sol
position = self[keccak256(abi.encodePacked(owner, tickLower, tickUpper))];
```

```
// contracts/interfaces/pool/IUniswapV3PoolState.sol
/// @param key The position's key is a hash of a preimage composed by the owner, tickLower
and tickUpper
function positions(bytes32 key) external view returns (...);
```

Impact:

Off-chain tooling / integrations may compute incorrect keys and fail to read positions correctly, leading to operational errors.

Recommendation:

Update NatSpec to explicitly specify `keccak256(abi.encodePacked(owner, tickLower, tickUpper))` (including the exact types) as the key derivation.

INFO

SqrtPriceMath.getAmount1Delta accepts sqrt ratios of 0 (inconsistent with getAmount0Delta)

Location:

`contracts/libraries/SqrtPriceMath.sol:175-194`

Description:

`SqrtPriceMath.getAmount0Delta()` explicitly requires the lower sqrt ratio to be non-zero (`require(sqrtRatioAX96 > 0)`), preventing invalid sqrt-price inputs that would break the formula.

By contrast, `getAmount1Delta()` performs no analogous validation and will happily compute an amount when one or both sqrt ratios are `0`.

Although Uniswap V3 pool logic typically never supplies `0` sqrt ratios (they come from `TickMath` and the pool's `sqrtPriceX96`), the library API is inconsistent and can silently produce results for invalid inputs if reused elsewhere.

Evidence:

```
function getAmount1Delta(
 uint160 sqrtRatioAX96,
 uint160 sqrtRatioBX96,
 uint128 liquidity,
 bool roundUp
) internal pure returns (uint256 amount1) {
 if (sqrtRatioAX96 > sqrtRatioBX96) (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96,
sqrtRatioAX96);

 return
 roundUp
 ? FullMath.mulDivRoundingUp(liquidity, sqrtRatioBX96 - sqrtRatioAX96,
FixedPoint96.Q96)
 : FullMath.mulDiv(liquidity, sqrtRatioBX96 - sqrtRatioAX96, FixedPoint96.Q96);
}
```

Impact:

If any downstream code path ever passes `sqrtRatioAX96 == 0` (e.g., uninitialized/invalid state), `getAmount1Delta()` will not fail fast and can return misleading amounts, potentially causing incorrect token accounting.

Recommendation:
Consider validating that both sqrt ratios are within the expected non-zero Uniswap range (at least `> 0`, ideally within `[TickMath.MIN_SQRT_RATIO, TickMath.MAX_SQRT_RATIO)`) or clearly document that callers must provide valid sqrt ratios.

INFO

SqrtPriceMath: documented input/bounds checks are incomplete (can return out-of-tick-range prices or divide by zero if mis-used)

Locations:

`contracts/libraries/SqrtPriceMath.sol:28-56`

`contracts/libraries/SqrtPriceMath.sol:58-97`

`contracts/libraries/SqrtPriceMath.sol:99-143`

Description:

Several `SqrtPriceMath` functions are documented as validating inputs / bounds ("throws if price or liquidity are 0, or if the next price is out of bounds"), but the actual checks are incomplete:

1) The low-level helpers `getNextSqrtPriceFromAmount0RoundingUp()` and `getNextSqrtPriceFromAmount1RoundingDown()` do **not** validate `sqrtPX96 > 0` and `liquidity > 0`. If called directly with `sqrtPX96 == 0` or `liquidity == 0`, they can:

- revert due to division-by-zero (`numerator1 / sqrtPX96`, or `/ liquidity`), or
- return an invalid next price of `0` (e.g., `mulDivRoundingUp(..., sqrtPX96=0, ...)` returns `0`).

2) The wrappers `getNextSqrtPriceFromInput()` / `getNextSqrtPriceFromOutput()` only validate `sqrtPX96 > 0` and `liquidity > 0`. They **do not** enforce that the computed `sqrtQX96` remains within Uniswap's tick-price bounds (`[TickMath.MIN_SQRT_RATIO, TickMath.MAX_SQRT_RATIO)`). It is possible (mathematically) for these functions to return a `uint160` value that is still within `uint160` range but outside the protocol's valid sqrt-price range, contrary to the docstring.

In Uniswap V3 Core's current swap path this is usually mitigated by higher-level clamping to the swap target price/limit, but the library itself does not enforce what it claims.

Evidence:

Missing non-zero checks in low-level functions:

```
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;
 ...
 return uint160(UnsafeMath.divRoundingUp(numerator1, (numerator1 / sqrtPX96).add(amount)-
));
}

function getNextSqrtPriceFromAmount1RoundingDown(
 uint160 sqrtPX96,
 uint128 liquidity,
 uint256 amount,
 bool add
) internal pure returns (uint160) {
 uint256 quotient = (amount << FixedPoint96.RESOLUTION) / liquidity; // no liquidity>0
check
 ...
}
```

Wrapper docstrings promise “out of bounds” checks but don’t implement them:

```
/// @dev Throws if price or liquidity are 0, or if the next price is out of bounds
function getNextSqrtPriceFromInput(...) internal pure returns (uint160) {
 require(sqrtPX96 > 0);
 require(liquidity > 0);
 return ...;
}
```

Impact:

- If these helpers are called directly (or reused in other contexts) with invalid `sqrtPX96`/`liquidity`, they can revert unexpectedly (division-by-zero) or return `sqrtQX96 == 0`.
- Even with `sqrtPX96 > 0` and `liquidity > 0`, callers that rely on the wrappers’ docstrings may assume `sqrtQX96` is always within `[MIN_SQRT_RATIO, MAX_SQRT_RATIO)`, but it is not explicitly enforced here. Downstream consumers (e.g., `TickMath.getTickAtSqrtRatio`) will revert if fed an out-of-range value.

Recommendation:

- Either (a) enforce the documented preconditions/bounds in these functions (including TickMath min/max sqrt-price range), or (b) update the documentation to clearly state that bounds are enforced only by higher-level logic and that direct callers must validate inputs and clamp outputs.

INFO

Tick library functions do not validate tick index is within TickMath bounds

Location:

`contracts/libraries/Tick.sol:60-185`

Description:

`Tick.getFeeGrowthInside()`, `Tick.update()`, `Tick.clear()`, and `Tick.cross()` accept arbitrary `int24 tick`/`tickLower`/`tickUpper` values and do not enforce that these tick indices are within the canonical Uniswap bounds `[TickMath.MIN_TICK, TickMath.MAX_TICK]`.

In the canonical Uniswap V3 pool flow, the pool contract performs these validations (e.g., via `checkTicks()`), so this is usually safe. But the lack of validation in the library itself makes it easy for future refactors or alternative integrations to accidentally write/read tick state for out-of-range ticks, which will later fail when interacting with `TickMath` or tick bitmap logic.

Evidence:

For example, `update()` writes to `self[tick]` without bounds checks:

```
function update(
 mapping(int24 => Tick.Info) storage self,
 int24 tick,
 int24 tickCurrent,
 int128 liquidityDelta,
 ...
) internal returns (bool flipped) {
 Tick.Info storage info = self[tick];
 ...
}
```

Impact:

If any call path ever bypasses higher-level tick validation, tick state can be created for invalid ticks, potentially leading to:

- downstream reverts when computing prices (`TickMath.getSqrtRatioAtTick`) or finding next ticks
- inconsistent accounting / inability to cross or clear such ticks

Recommendation:

Consider asserting tick bounds inside the library on write paths (`update`, `cross-`, `clear`) or clearly documenting that callers MUST validate tick indices against `Tick-Math` bounds.

INFO

Tick.Info.initialized is not inherently equivalent to liquidityGross != 0 (relies on external clear), contrary to comment

Locations:

`contracts/libraries/Tick.sol:34-37`

`contracts/libraries/Tick.sol:110-150`

`contracts/libraries/Tick.sol:152-157`

Description:

`Tick.Info.initialized` is documented as being "exactly equivalent" to `liquidityGross != 0`, but `Tick.update()` **never sets** `initialized` back to `false` when liquidity goes to zero.

Instead, correctness depends on the external caller remembering to invoke `Tick.clear()` (which deletes the struct and resets `initialized`). This is a subtle business-logic invariant that can be broken by caller changes/refactors.

Evidence:

Comment asserting equivalence:

```
// true iff the tick is initialized, i.e. the value is exactly equivalent to the expression
liquidityGross != 0
bool initialized;
```

(See `contracts/libraries/Tick.sol:34-37`.)

But `update()` only ever sets it `true`:

```
if (liquidityGrossBefore == 0) {
 ...
 info.initialized = true;
}
```

(See `contracts/libraries/Tick.sol:132-142`.)

And the only way to reset is deleting the entry:

```
function clear(mapping(int24 => Tick.Info) storage self, int24 tick) internal {
 delete self[tick];
}
```

(See `contracts/libraries/Tick.sol:152-157`.)

Impact:

- If a caller forgets to `clear()` after `liquidityGross` reaches 0, the tick will remain marked `initialized == true` with `liquidityGross == 0`, violating the documented invariant.
- This can cause other logic that checks only the `initialized` flag (e.g., snapshot validation patterns) to treat a tick as initialized and use stale/meaningless accumulator values, leading to incorrect results or unexpected behavior.

Recommendation:

Either (a) update the comment to reflect the dependency on `clear()` and make the invariant explicit, or (b) set `initialized = false` when `liquidityGrossAfter == 0` inside `update()` (and still delete/clear as an optimization if desired).

INFO

Tick.cross lacks state precondition check (can be called on uninitialized ticks and corrupt tick accounting)

Location:

`contracts/libraries/Tick.sol:168-184`

Description:

`Tick.cross()` mutates a tick's `feeGrowthOutside*`, `secondsPerLiquidityOutsideX128`, `tickCumulativeOutside`, and `secondsOutside` by subtracting stored values from globals, and returns `liquidityNet`.

However, it does **not** validate that the tick is actually initialized (i.e., `info.initialized == true` / `liquidityGross != 0`). If `cross()` is called on an uninitialized tick (default-zero struct), it will still write non-zero "outside" values derived from the current globals and timestamp.

In Uniswap V3's canonical pool flow, `cross()` is only called for initialized ticks discovered via the tick bitmap, so this typically won't trigger. But as a library function, the missing state precondition is an input-validation gap: misuse can permanently pollute tick state and break fee/seconds accounting.

Evidence:

```
function cross(
 mapping(int24 => Tick.Info) storage self,
 int24 tick,
 uint256 feeGrowthGlobal0X128,
 uint256 feeGrowthGlobal1X128,
 uint160 secondsPerLiquidityCumulativeX128,
 int56 tickCumulative,
 uint32 time
) internal returns (int128 liquidityNet) {
 Tick.Info storage info = self[tick];
 info.feeGrowthOutside0X128 = feeGrowthGlobal0X128 - info.feeGrowthOutside0X128;
 info.feeGrowthOutside1X128 = feeGrowthGlobal1X128 - info.feeGrowthOutside1X128;
 ...
 info.secondsOutside = time - info.secondsOutside;
 liquidityNet = info.liquidityNet;
}
```

No `require(info.initialized)` or equivalent is present.

Impact:

If a call path ever invokes `cross()` with an uninitialized tick (e.g., due to incorrect tick selection logic, alternative integration, or refactor), the tick's accounting fields can be set to values that do not correspond to any real liquidity boundary. This can lead to incorrect fee growth calculations and broken oracle/seconds accounting for positions that later use that tick.

Recommendation:

Add an explicit state precondition (e.g., `require(info.initialized)` / `require(info.liquidityGross != 0)`) or clearly document that callers must only call `cross()` for initialized ticks.

INFO

TickBitmap relies on external tickSpacing constraints; tickSpacing=0/negative or too large can revert or overflow int24 math

Location:

`contracts/libraries/TickBitmap.sol:23-76`

Description:

`TickBitmap.flipTick()` and `TickBitmap.nextInitializedTickWithinOneWord()` assume `tickSpacing` is a **positive, non-zero, and sufficiently small** integer.

The library:

- performs `tick % tickSpacing` and `tick / tickSpacing` without checking `tickSpacing != 0`, which will revert on division/modulo by zero;
- uses `int24`-typed arithmetic like `(compressed ± something) * tickSpacing` which can **wrap in Solidity <0.8** if `tickSpacing` is too large, returning an incorrect `next` tick that may not be caught by downstream clamping.

The canonical `UniswapV3Factory.enableFeeAmount()` enforces `tickSpacing > 0 && tickSpacing < 16384`, which prevents these issues in standard deployments, but the library itself is not self-contained and is unsafe if reused.

Code Snippet:

```
require(tick % tickSpacing == 0); // tickSpacing==0 => revert
(int16 wordPos, uint8 bitPos) = position(tick / tickSpacing);
...
int24 compressed = tick / tickSpacing;
...
next = ... * tickSpacing; // potential int24 wrap if tickSpacing is large
```

Impact:

If a caller supplies an invalid `tickSpacing` (0/negative/too large), tick search and bitmap flips can revert or compute incorrect ticks due to overflow/wraparound. In a swap loop, this can cause incorrect tick traversal, missed initialized ticks, or unexpected behavior.

Recommendation:

Enforce `tickSpacing > 0` (and optionally an upper bound) at library boundaries or document the required invariants prominently and ensure all call sites validate them.

INFO

TickBitmap.nextInitializedTickWithinOneWord NatSpec likely misstates distance as "256 ticks" (actually 256 compressed ticks => 256*tickSpacing)

Location:

`contracts/libraries/TickBitmap.sol:34-42`

Description:

`TickBitmap.nextInitializedTickWithinOneWord` searches within a single 256-bit word of the **compressed tick index** (tick / tickSpacing). The NatSpec return description says the result is “up to 256 ticks away from the current tick”, which is ambiguous/misleading when `tickSpacing > 1`.

In reality, the search window is **256 compressed ticks**, so the returned `next` can be up to `256 * tickSpacing` in raw tick units.

Evidence:

```
/// @return next The next initialized or uninitialized tick up to 256 ticks away from the
current tick
...
int24 compressed = tick / tickSpacing;
...
next = ... * tickSpacing;
```

Impact:

Off-chain and integrator logic that relies on the comment may misestimate search bounds and incorrectly reason about worst-case tick movement.

Recommendation:

Clarify the NatSpec to explicitly state the function searches within one 256-bit word of the **compressed** tick bitmap (i.e., up to `256 * tickSpacing` in raw ticks).

INFO

TickMath.getTickAtSqrtRatio documentation and endpoint semantics are inconsistent ("getRatioAtTick" reference and MAX_SQRT_RATIO inverse failure)

Locations:

`contracts/libraries/TickMath.sol:56-64`

`contracts/libraries/TickMath.sol:13-16`

`contracts/libraries/TickMath.sol:48-54`

Description:

`TickMath.getTickAtSqrtRatio()` contains documentation that does not match the implementation and can mislead integrators:

1) The doc refers to a non-existent/incorrect function and variable:

- "greatest tick value such that `getRatioAtTick(tick) <= ratio`"
- "MIN_SQRT_RATIO is the lowest value `getRatioAtTick` may ever return"

…but this library's API is `getSqrtRatioAtTick()` and the input is `sqrtPriceX96`.

2) The library defines `MAX_SQRT_RATIO` as "the maximum value that can be returned from `getSqrtRatioAtTick`" (true), but `getTickAtSqrtRatio` **rejects** `sqrtPriceX96 == MAX_SQRT_RATIO` via a strict `< MAX_SQRT_RATIO` bound:

```
require(sqrtPriceX96 >= MIN_SQRT_RATIO && sqrtPriceX96 < MAX_SQRT_RATIO, 'R');
```

This means `getTickAtSqrtRatio(getSqrtRatioAtTick(MAX_TICK))` will revert, so the two functions are not mutual inverses at the upper endpoint. This also weakens the comment in `getSqrtRatioAtTick` claiming rounding "so getTickAtSqrtRatio of the output price is always consistent".

Impact:

Primarily a semantic/documentation mismatch and endpoint footgun:

- Developers may assume `getTickAtSqrtRatio` can accept any output of `getSqrtRatioAtTick`, including the maximum.

- Off-chain/integration code may inadvertently pass `MAX_SQRT_RATIO` and hit unexpected reverts.

Core Uniswap V3 pool logic avoids using `MAX_SQRT_RATIO` directly (typically using `MAX_SQRT_RATIO - 1`), but the library itself doesn't communicate this clearly.

Recommendation:
Align the documentation with actual behavior (reference `getSqrtRatioAtTick`/`sqrtPriceX96`, and explicitly document the strict `< MAX_SQRT_RATIO` requirement and the non-invertible upper endpoint).

INFO

Uniswap V3 oracle is economically manipulable for short windows / early after initialization (one observation per block)

Locations:

```
contracts/UniswapV3Pool.sol:235-252
contracts/UniswapV3Pool.sol:255-267
contracts/UniswapV3Pool.sol:269-289
contracts/libraries/Oracle.sol:47-63
contracts/libraries/Oracle.sol:65-101
contracts/libraries/Oracle.sol:245-258
```

Description:

The pool's built-in oracle (`observe()` / `snapshotCumulativesInside()`) is designed for TWAP use, but it has properties that make it economically manipulable when external consumers use **short lookback windows**, **newly initialized pools**, or **low-liquidity pools**.

Key design points:

- Pools initialize the oracle with a single "anchor" observation containing zeroed cumulatives.
- Oracle observations are writable **at most once per block**; additional swaps in the same block do not create more oracle points.
- For `secondsAgo == 0`, the oracle may synthesize a counterfactual observation for the current block using the **current tick**, making the instantaneous "price" easy to influence via same-block swaps.

These behaviors are not necessarily bugs in Uniswap V3 itself, but they are a frequent source of economic exploits against external protocols that treat Uniswap V3's `observe()` as a manipulation-resistant oracle without imposing minimum-age/liquidity/window constraints.

Evidence:

Oracle starts with a single zeroed observation:

```
// contracts/libraries/Oracle.sol
function initialize(Observation[65535] storage self, uint32 time)
 internal
 returns (uint16 cardinality, uint16 cardinalityNext)
{
 self[0] = Observation({
 blockTimestamp: time,
 tickCumulative: 0,
 secondsPerLiquidityCumulativeX128: 0,
 initialized: true
 });
 return (1, 1);
}
```

At most one observation per block:

```
// contracts/libraries/Oracle.sol
// early return if we've already written an observation this block
if (last.blockTimestamp == blockTimestamp) return (index, cardinality);
```

secondsAgo==0 can use current tick (counterfactual transform):

```
// contracts/libraries/Oracle.sol
if (secondsAgo == 0) {
 Observation memory last = self[index];
 if (last.blockTimestamp != time) last = transform(last, time, tick, liquidity);
 return (last.tickCumulative, last.secondsPerLiquidityCumulativeX128);
}
```

Pool exposes observe()/snapshot functionality to consumers:

```
// contracts/UniswapV3Pool.sol
function observe(uint32[] calldata secondsAgos) external view override noDelegateCall
 returns (int56[] memory tickCumulatives, uint160[] memory secondsPerLiquidityCumula-
tiveX128s)
{ ... }
```

Impact:

- **External-oracle exploitation risk:** Protocols that use short TWAP windows (or `secondsAgo==0`) can have their price feed manipulated within a single block using swaps/flash liquidity.
- **New pool fragility:** Immediately after `initialize()`, there is minimal oracle history; early TWAPs can be dominated by attacker-controlled initial state.

Recommendation:

- External consumers should enforce:
  - sufficiently long TWAP windows,
  - minimum pool liquidity,
  - minimum pool age / observation cardinality,
  - or multi-source oracle aggregation.
- If a deployment intends Uniswap pools to serve as strong oracles, operationally increase `observationCardinalityNext` early and discourage/avoid using newly initialized pools as price sources.

INFO

UnsafeMath.divRoundingUp returns 0 on division-by-zero (EVM semantics), which can silently corrupt amount/price math if callers fail to validate denominators

Location:

`contracts/libraries/UnsafeMath.sol:7-16`

Description:

`UnsafeMath.divRoundingUp` is implemented in assembly and explicitly does not check for `y == 0`:

```
// division by 0 has unspecified behavior, and must be checked externally
assembly {
 z := add(div(x, y), gt(mod(x, y), 0))
}
```

On EVM, `div(x, 0)` and `mod(x, 0)` evaluate to 0, so `divRoundingUp(x, 0)` returns 0 **without reverting**.

Impact:

If any caller mistakenly passes `y == 0`, computations that rely on rounding-up division may silently produce 0, potentially:

- Miscomputing next price steps.
- Miscomputing token amount deltas.

In asset-handling contexts, this can translate into **incorrect amounts owed/paid**, potentially trapping assets or enabling value leakage.

Recommendation:

Ensure all call sites validate denominators are non-zero before calling. Where practical, prefer a checked version that reverts on `y == 0` for safer integration.

INFO

`observe()` NatSpec uses incorrect log base for tick (sqrt(1.0001) vs 1.0001)

Location:

`contracts/interfaces/pool/IUniswapV3PoolDerivedState.sol:8-14`

Description:

`IUniswapV3PoolDerivedState.observe()` describes the time-weighted average tick as being in “log base sqrt(1.0001)”, which is inconsistent with the rest of the codebase and with the `Initialize` event docs stating tick is “log base 1.0001”. Using the wrong base would scale the tick by a factor of 2, leading to incorrect price reconstruction.

Evidence:

```
// contracts/interfaces/pool/IUniswapV3PoolDerivedState.sol
/// @dev The time weighted average tick represents the geometric time weighted average price
of the pool, in
/// log base sqrt(1.0001) of token1 / token0.
```

Contrasting doc in the same interface set:

```
// contracts/interfaces/pool/IUniswapV3PoolEvents.sol
/// @param tick The initial tick of the pool, i.e. log base 1.0001 of the starting price of
the pool
```

Impact:

Off-chain code that follows this NatSpec literally may compute incorrect TWAP prices (systematic factor-of-2 error in tick/log space), which is a numerical correctness issue for oracles, analytics, and risk checks.

Recommendation:

Correct the NatSpec to “log base 1.0001 of token1/token0” (consistent with how `TickMath` interprets ticks).

INFO

slot0() NatSpec references non-existent SqrtTickMath library (should be TickMath)

Location:

`contracts/interfaces/pool/IUniswapV3PoolState.sol:8-20`

Description:

`IUniswapV3PoolState.slot0()` NatSpec claims `tick` may differ from `SqrtTickMath.getTickAtSqrtRatio(sqrtPriceX96)`, but the canonical library/function is `TickMath.getTickAtSqrtRatio` (there is no `SqrtTickMath` in this codebase).

This is minor but can mislead integrators implementing oracle/tick math.

Evidence:

```
/// This value may not always be equal to SqrtTickMath.getTickAtSqrtRatio(sqrtPriceX96) if
the price is on a tick
/// boundary.
function slot0() external view returns (..., int24 tick, ...);
```

Impact:

Documentation mismatch can cause confusion and implementation mistakes for external consumers computing ticks/prices.

Recommendation:

Update NatSpec to reference `TickMath.getTickAtSqrtRatio(sqrtPriceX96)`.

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