

Code Assessment of the MetaDEX Slipstream Smart Contracts

October 21, 2026

Produced for

DROMOS LABS



Contents

1	Executive Summary	3
2	Assessment Overview	4
3	System Overview	6
4	Trust Model	10
5	System Considerations	13
6	Terminology	17
7	Findings	18
8	Limitations and use of report	38



1 Executive Summary

Dear Dromos Labs Team,

Thank you for trusting ChainSecurity with the review of the MetaDEX Slipstream. As a globally trusted security partner for tier-one DeFi applications, ChainSecurity provides specialized, independent analysis for complex smart contract systems. We greatly appreciate the opportunity to collaborate with Dromos Labs, the core contributing development company behind Aerodrome, the leading decentralized exchange on Base. MetaDEX Slipstream evolves the Aerodrome design and brings it to a broader set of chains, including Ethereum mainnet and Arbitrum-style networks.

Dromos Labs implements Slipstream, a concentrated-liquidity AMM forked from Uniswap V3. It tracks staked liquidity per tick, so that positions staked in a gauge earn emissions instead of swap fees, and lets a pluggable hook set the swap fee. We have conducted a thorough review of the scope with a focus on privileged roles, liveness, and the handling of failing external calls:

The gauge factory splits control over emission caps, referrals and penalties between several roles and the emergency council of the `LeafVoter`. The emergency council could not zero a gauge's emission cap at all, which has been resolved, see [setEmissionCap Reverts: LeafVoter Has No emergencyCouncil](#).

Staked positions must stay withdrawable at all times. A chain suspension locked staked NFTs with pending rewards, which has been resolved, see [Chain Suspension Locks Staked NFTs with Pending Rewards](#).

Pools wrap their calls into the swap hook and the fee modules so that a failing callee does not block swaps. A caller could starve these calls of gas to select their failure branch, see [Griefing via Passing Insufficient Gas](#), and the fixed gas stipends could not adapt to future gas repricing, see [Hardcoded Gas Stipends Cannot Adapt to Gas Repricing](#). The gas stipends have been removed. The swap hook is now called directly, so a failing hook reverts the swap or flash instead of selecting a fallback fee. The unstaked fee module is still called in a `try / catch` that forwards 63/64 of the available gas, so a malicious caller may still be able to force its fallback.

We provide this audit report to assist in forming an opinion on the risk profile of MetaDEX Slipstream. In summary, we find that the codebase provides a good level of security, if operated under the assumptions documented in [System Overview](#), [Trust Model](#) and [System Considerations](#).

The emergency withdrawal mechanism added in `Version 5`, which bypasses accounting that the rest of the system relies on, was reviewed on a short notice of only days before launch. One related issue was identified, see [Emergency Withdrawals Let New Stakes Capture Past Emissions](#). Due to the higher risk of undiscovered issues, further review and testing should be conducted before emergency withdrawals are relied on in production.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,
ChainSecurity

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the MetaDEX Slipstream repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Version	Date	Commit Hash	Note
1	17 June 2026	17330331b911fca44e5c8535b601778d7f7f3b48	Initial Version
2	10 August 2026	4d013bad41be21f1933c73a5c7ab9856baa624aa	Extended Scope
3	13 October 2026	7618b0a8ff5c0bf68b3dcf3aff246d16e8be6e96	Fix Review
4	19 October 2026	c7c7e95c352e8da9c0cc6e2e012f1fe5bb906aaa	Fixes Review
5	21 October 2026	cf1ae46f81bc45d5c0b7d15d22ef8c84eb777403	Final Version

For the Solidity smart contracts, the compiler version `0.7.6` was chosen.

The following files were reviewed in full:

```
contracts/
├── core/
│   ├── indexation/
│   │   └── CLFactoryIndexation.sol
│   └── libraries/
│       └── hook/
│           └── SwapHookLib.sol
├── gauge/
│   ├── CLGauge.sol
│   ├── CLGaugeFactory.sol
│   └── Gauge.sol
└── staker/
    ├── SlipstreamStaker.sol
    └── libraries/
        ├── IncentiveId.sol
        ├── NFTPositionInfo.sol
        ├── RewardMath.sol
        └── TransferHelperExtended.sol
```

In **Version 3**, `SlipstreamStaker` and its libraries were removed from the scope:

```
contracts/
└── staker/
    └── libraries/
        ├── IncentiveId.sol
        ├── NFTPositionInfo.sol
        └── RewardMath.sol
```

```
└─ TransferHelperExtended.sol
```

Further, the following files were added to the scope:

```
contracts/
├─ core/
│   └─ libraries/
│       └─ CLPoolNonSwapPath.sol
├─ gauge/
│   ├── extensions/
│   │   ├── ArbCLGauge.sol
│   │   └─ ArbCLGaugeFactory.sol
│   ├── libraries/
│   │   └─ Roles.sol
│   └─ tape/
│       └─ CLGaugeTape.sol
├─ libraries/
│   ├── ProtocolConstants.sol
│   └─ ProtocolTimeLibrary.sol
└─ periphery/
    └─ LpMigrator.sol
```

The files below were reviewed as a *diff* audit, meaning only the changes relative to the reference commit `479f512de8c72b863168d5b46788d21618d4e59f` were evaluated:

```
contracts/
├─ core/
│   ├── CLFactory.sol
│   └─ CLPool.sol
└─ periphery/
    └─ NonfungiblePositionManager.sol
```

2.1.1 Excluded from scope

All other contracts not explicitly mentioned are outside the scope of this review.

For the diff-audited files, we assume that the Slipstream codebase at the reference commit is correct and secure.

Further, the economic design and the correct choice of parameters are out of scope of this review.

The contracts are expected to be deployed on the Optimism and Base L2s. From **Version 3** on, they are also expected to be deployed on Arbitrum, Robinhood Chain, Ethereum Mainnet, Arc and Ink. Other chains are not covered by this review, and the differences in their behavior must be carefully considered before deploying there.

3 System Overview

This system overview describes the most recent version (`Version 3`) of the contracts.

Slipstream is Aerodrome's concentrated-liquidity AMM. The core pool and swap logic is a fork of Uniswap V3 (`v3-core` / `v3-periphery`), renamed from `UniswapV3*` to `CL*` but keeping the original callback names (`uniswapV3SwapCallback` , `uniswapV3MintCallback` , `uniswapV3FlashCallback`) for integrator compatibility. On top of this, the repository adds a gauge system that lets staked positions earn AERO emissions instead of swap fees.

The codebase splits into two main parts:

- **Pools:** liquidity, swaps, fee accounting, NFT positions. Upstream Uniswap V3 code plus Aerodrome-specific hooks.
- **Gauges:** emissions distribution to staked positions and routing of their fees to voters. Aerodrome-native.

3.1 Pools

`CLFactory` deploys pools as EIP-1167 clones of a single `CLPool` implementation, deterministically salted on (`token0` , `token1` , `tickSpacing`). Only tick spacings explicitly enabled by the factory owner can be used. The initial set is passed to the constructor. The default deployment configuration enables `1 / 50 / 100 / 200 / 2000` with default fees of 1/5/5/30/100 bps respectively (fees are expressed in pips, i.e. millionths, matching the Uniswap V3 convention rather than Aerodrome's usual bps). Pools are created independently of gauges and always come initialized with a starting price. After deployment, each pool is registered in the `factoryRegistry` via `registerTarget` , so a gauge can be created for it immediately.

The factory accepts a `voter` parameter in its constructor. If the factory is deployed with `voter == address(0)` , the `setVoter()` setter allows it to be set exactly once after deployment. This setter is callable only by the factory owner, and only while `voter` remains the zero address. The `factoryRegistry` is supplied directly as a constructor argument.

`CLPool` extends the standard Uniswap V3 tick/liquidity/oracle model with a second, per-tick liquidity value: **staked liquidity**. When a position is staked into its gauge, the pool moves the position's liquidity from being owned by the `NonfungiblePositionManager` to being owned virtually by the gauge, while keeping it in the same tick range.

This lets the pool track, per tick, how much of the active liquidity belongs to staked (gauge) positions versus unstaked (fee-earning) positions, and split swap/flash fees between the two accordingly. A configurable **unstaked fee** is levied on the portion of fees earned by unstaked LPers. The rest of staked-position fees are collected in bulk by the gauge. Flash loan fees accrue only to the liquidity in range at the time of the loan.

`CLPool` exposes a `POOL_TYPE` constant (`bytes32("CL")`) used by the meta-router for dispatch. The pool also provides three fee views: `beforeSwapFee(SwapParams)` and `afterSwapFee(SwapParams, AfterSwapParams)` return the fee applied at each stage of the swap loop, and `flashFee(FlashParams)` returns the flash loan fee. A `fee()` view is retained for backward compatibility and delegates to `beforeSwapFee` with `msg.sender` as the caller, so discount lookups continue to work through it.

3.2 Fee modules

`CLPool` does not hard-code its swap fee. The swap fee is determined by a pluggable swap hook set on the factory. Each pool has an associated hook address, defaulting to the factory-wide hook unless overridden per pool. The hook implements the `ISwapHook` interface and is invoked by `SwapHookLib` at the start and end of each swap and at the start of each flash loan.

If the hook returns a fee above `_SWAP_FEE_CEIL` (10%), the fee is clamped to the ceiling. If the hook is unset, or its `beforeSwap()` returns a zero fee, the fee falls back to the pool's tick-spacing default. A reverting hook reverts the swap or flash. `SwapHookLib` also exposes three read-only helpers, `getBeforeSwapFee`, `getAfterSwapFee`, and `getFlashFee`, that back the corresponding fee views on `CLPool`.

The unstaked fee is controlled separately via a pluggable `unstakedFeeModule` set by the `unstakedFeeManager`. If no module is configured, the factory applies a default unstaked fee of 10%. No unstaked fee is charged while the pool has no gauge, its gauge is deactivated, or its emission cap is zero.

All fee roles (`swapFeeManager`, `unstakedFeeManager`) are independent, factory-owner-appointed addresses, distinct from the factory `owner` itself. The `swapFeeManager` controls per-pool hook assignments on `CLFactory` via `setDefaultSwapHook()` and `setPoolSwapHook()`.

3.3 Pool discovery

`CLFactoryIndexation` maintains append-only on-chain indexes covering all pools with creation timestamps, all tokens ever used, and pools grouped by token. These are populated automatically whenever `createPool` succeeds. They exist purely for discoverability (e.g. by off-chain indexers or other contracts) and are paginated to avoid unbounded reads.

3.4 Periphery

`NonfungiblePositionManager` is the standard entry point for LPs: it wraps pool positions as ERC-721 tokens and keeps separate fee accounting for staked and unstaked positions so that a staked position's owed fees stay at zero while staked.

`LpMigrator` is a permissionless helper that moves a position between position managers. It withdraws all liquidity and fees from the caller's NFT and burns it, optionally pulls additional tokens from the caller, and mints a new position. If no target pool is given, the pool is created at the source pool's price. At the end, the contract's entire token0 and token1 balance is sent to the caller.

3.5 Gauges

A gauge is created per pool by `CLGaugeFactory`, callable only by the external `gaugeManager` address. `createGauge()` deploys the gauge as an EIP-1167 clone and creates the associated `VotingRewardsManager` via `VotingRewardsFactory`. All state writes complete before any external call, following the checks-effects-interactions order. The CREATE2 salt is keyed on the pool address alone. `CLGauge.initialize()` requires `msg.sender == gaugeFactory` to prevent re-initialization by arbitrary callers. On Arbitrum, `ArbCLGaugeFactory` deploys `ArbCLGauge` clones instead, which measure deposit blocks and the early-unstake penalty window in Arbitrum L2 blocks (`arbBlockNumber()`).

Depositing an NFT into the gauge (`deposit`) settles any pending fees to the depositor, moves the position's liquidity to staked accounting in the pool, and starts emissions accrual. Withdrawing (`withdraw`) reverses this and auto-claims pending rewards. If minting the rewards reverts, they are recorded as `deferredEmissions` of the owner and the withdrawal proceeds. Deferred emissions are paid on the owner's next `claimEmissions()` call, which does not require a staked position. Withdrawal is always possible, even from a gauge that has been capped to zero emissions. While the factory has enabled emergency withdrawals, the stake owner can also call `emergencyWithdraw(tokenId)`. It works like `withdraw`, but skips the emission claim. Reward accrual uses a pull model driven by the `LeafVoter`. The voter maintains a monotonically increasing cumulative reward share for each gauge, accessible via `LeafVoter.settleGauge(gauge)`. This is a public, permissionless function that settles the voter's internal accounting and returns the running total. It returns zero for unregistered addresses without reverting.

`CLGauge` stores a `lastCumulativeRewardShare` cursor. Whenever the pool calls `_settleToBlock()`, it calls `gauge.settleGauge()`. The gauge calls `voter.settleGauge(address(this))`, takes the delta since its cursor, advances the cursor, and returns the amount as `_rewardsToDistribute`. The pool feeds this directly into the global reward-growth accumulator. A position's claimable emissions depend on how much of that growth occurred while its tick range was active.

When `setGaugeAndPositionManager()` first connects a gauge to a pool, the pool calls `gauge.settleGauge()` immediately. Any emissions the voter allocated before the connection are pulled at this point. Since no liquidity is staked yet, they accrue into rollover.

If the active tick has no staked liquidity, emissions accumulate as rollover in the pool. `CLGauge` exposes a public `forfeitRollover()` function. It calls `pool.settleToBlock()`, which returns and clears the rollover, then forwards it to `voter.forfeitEmissions()`. This lets anyone return idle emissions to the voter's surplus pool.

The `earned()` view estimates claimable rewards by calling `voter.projectedCumulativeRewardShare(address(this))`, a read-only projection at the current block, and diffing it against `lastCumulativeRewardShare`.

Two mechanisms sit on top of the base reward calculation, both configured on `CLGaugeFactory`:

- **Early-unstake penalty:** if a position's rewards are claimed before it has been staked for a minimum number of blocks, a configurable fraction of the reward is forfeited back to the Voter instead of being paid out. This discourages stake-claim-unstake cycling around reward distribution.
- **Referral fee:** a gauge can have a referral address and share (in pips) configured, which redirects part of every claim to the referrer. The referral share is credited to `deferredReferralEmissions` and minted when the referrer, or an operator it approved, calls `claimReferralEmissions()`.

Each gauge also has a per-second **emission cap**, set to the factory-wide default at gauge creation and adjustable per gauge afterwards. A cap of zero stops both new emissions and fee collection for that gauge. `CLGaugeFactory` gates these controls with OpenZeppelin `AccessControl` roles rather than a single owner: a cap admin (plus a narrower-range cap operator) controls emission caps, a referral admin controls referral configuration, a penalty admin controls the penalty rate and per-gauge minimum stake duration, a gauge tape manager, appointed by a gauge tape manager admin, sets the gauge tape and its gas limit, and an emergency withdraw manager, appointed by an emergency withdraw admin, enables and disables emergency withdrawals for all gauges of the factory. Holders of the `LeafVoter`'s `EMERGENCY_COUNCIL_ROLE` can set any gauge's cap to zero, including through `clearEmissionCap()`, which proceeds even if flushing the gauge's fees fails.

3.6 Gauge Tape

`CLGaugeTape` records each CL gauge's cumulative reward share and credited emissions for APR reads. On every settlement, the gauge reports to `CLGaugeFactory`, which forwards the data to the tape in a gas-capped call whose failure is ignored. Only factories whitelisted by the tape's owner may record. Observations are kept in an hourly ring of 48 slots and a daily ring of 32 slots.

3.7 Changes in Version 3

In `Version 3`, the following changes were made.

Gauges and emissions:

- The `DEFAULT_CAP_INDICATOR` sentinel was removed. A new gauge stores the factory's `defaultCap` at creation, so later changes of the default only affect gauges created afterwards. The `[operatorMinCap, operatorMaxCap]` range of `CAP_OPERATOR_ROLE` now always applies, and the operator cannot change a gauge whose cap is zero.

- The emergency council is the `EMERGENCY_COUNCIL_ROLE` on `LeafVoter` instead of a single `emergencyCouncil` address. `clearEmissionCap` was added, which zeroes a gauge's cap even if flushing its fees fails. `setEmissionCaps` sets several caps at once.
- If minting fails during a withdrawal, the emissions are kept in `deferredEmissions` and the withdrawal proceeds. `claimEmissions()` pays them without requiring a staked position.
- Referral emissions are no longer minted at claim time. They are credited to `deferredReferralEmissions` and minted through `claimReferralEmissions()`.
- `CLGaugeTape` was added, together with `GAUGE_TAPE_MANAGER_ROLE` and `GAUGE_TAPE_MANAGER_ADMIN_ROLE` on `CLGaugeFactory`.
- `ArbCLGaugeFactory` and `ArbCLGauge` were added for Arbitrum. They measure deposit blocks and the penalty window in Arbitrum L2 blocks.

Pools:

- No unstaked fee is charged while the pool has no gauge, its gauge is deactivated, or its emission cap is zero.
- The initial fee tiers are passed to the `CLFactory` constructor instead of being hard-coded.

Periphery and staker:

- `LpMigrator` was added.
- `SlipstreamStaker` and its libraries were removed from the codebase. It was a permissionless contract, modeled on Uniswap's `v3-staker`, that let anyone fund time-boxed reward campaigns on any pool, in which LPs staked their position NFTs separately from the gauges.

3.8 Changes in Version 5

In `Version 5`, the following changes were made.

- `CLGauge.emergencyWithdraw()` was added. While emergency withdrawals are enabled on the `CLGaugeFactory`, it returns the caller's staked position without any emission accounting. `CLGaugeFactory` gained `EMERGENCY_WITHDRAW_ROLE` to enable and disable emergency withdrawals, administered by `EMERGENCY_WITHDRAW_ADMIN_ROLE`.

4 Trust Model

No contract in scope uses a proxy pattern. `CLPool`, `CLGauge` and `ArbCLGauge` are EIP-1167 minimal clones of immutable implementation contracts. `CLGaugeFactory`, `ArbCLGaugeFactory`, `NonfungiblePositionManager`, and `LpMigrator` are all non-upgradeable.

4.1 Roles

CLGaugeFactory - CAP_ADMIN_ROLE

- Trust level: Partially trusted.
- Permissions: Sets the nonzero default emission cap applied to newly created gauges, and the per-gauge emission cap for any registered gauge to any value, including zero. Also sets the `[operatorMinCap, operatorMaxCap]` range that bounds `CAP_OPERATOR_ROLE`.
- Worst-case impact: Setting any gauge's cap to zero stops new emission accrual and prevents the gauge from collecting fees via `collectFees`. The extra unstaked fee stops being charged, but the share of each swap and flash fee attributable to staked liquidity keeps accruing to the pool's `gaugeFees` slot, and the gauge refuses to collect it while its cap is zero. The same holds while the gauge is deactivated in the Voter. These fees remain locked until the cap is raised or the gauge is reactivated. Changing the default cap does not affect existing gauges. The cap admin cannot steal LP principal or redirect already-accrued rewards.

CLGaugeFactory - CAP_OPERATOR_ROLE

- Trust level: Partially trusted.
- Permissions: Can update per-gauge emission caps to values within `[operatorMinCap, operatorMaxCap]` as set by `CAP_ADMIN_ROLE`. Cannot change the cap of a gauge whose cap is zero. Cannot set a cap to zero, since `operatorMinCap` must be nonzero.
- Worst-case impact: Can raise or lower emission throughput for individual gauges within the operator range, distorting rewards across pools. Cannot disable a gauge or redirect LP funds.

CLGaugeFactory - REFERRAL_ADMIN_ROLE

- Trust level: Partially trusted.
- Permissions: Sets the per-gauge referral address and referral share (in pips, capped at `maxShareCap`, default 5%). Also sets `maxShareCap` itself (capped at 100%).
- Worst-case impact: Can redirect up to `maxShareCap` of every claim's gross emissions from any gauge to an arbitrary address. Changes apply to the next claim or withdrawal and to the entire un-checkpointed accrual of a position. The role is its own role admin, so no other role can revoke it. If `maxShareCap` is first raised to 100% and then the referral share for a gauge is set to 100%, all future emission claims from that gauge are diverted entirely to the referral address. Cannot affect LP principal or rewards already accrued inside the pool.

CLGaugeFactory - PENALTY_ADMIN_ROLE

- Trust level: Partially trusted.
- Permissions: Sets the global early-exit penalty rate (in pips, up to 100%) and the global minimum stake duration (in blocks, bounded by the constructor-immutable `maxMinStakeBlocks`). Can also set a per-gauge minimum stake duration override.
- Worst-case impact: Setting the penalty rate to 100% and the minimum stake window to `maxMinStakeBlocks` causes any LP that claims before that window expires to forfeit their entire accrued reward back to the Voter. Changes apply to the entire un-checkpointed accrual of a position. Since `depositBlock` is only reset on deposit, raising the window also subjects positions that have been staked

longer than the previous window to the penalty. Cannot affect LP principal held in the NFT, which is always withdrawable regardless of penalty configuration.

CLGaugeFactory - GAUGE_TAPE_MANAGER_ADMIN_ROLE

- Trust level: Partially trusted.
- Permissions: Grants and revokes `GAUGE_TAPE_MANAGER_ROLE`. The role is its own role admin.
- Worst-case impact: Can appoint a malicious gauge tape manager and therefore carries the `GAUGE_TAPE_MANAGER_ROLE` worst case.

CLGaugeFactory - GAUGE_TAPE_MANAGER_ROLE

- Trust level: Partially trusted.
- Permissions: Sets the gauge tape address and its relay gas limit via `setClGaugeTape`. The gas limit is only checked to be nonzero.
- Worst-case impact: A gas limit close to the block gas limit makes every `settleGauge` relay revert. Gauge deposits, withdrawals, claims and tick-crossing swaps then revert, so staked NFTs cannot be withdrawn until the manager lowers the limit or clears the tape address.

CLGaugeFactory - EMERGENCY_WITHDRAW_ADMIN_ROLE

- Trust level: Fully trusted.
- Permissions: Grants and revokes `EMERGENCY_WITHDRAW_ROLE`. The role is its own role admin.
- Worst-case impact: Can appoint a malicious emergency withdraw manager and therefore carries the `EMERGENCY_WITHDRAW_ROLE` worst case. If its last holder renounces it, no emergency withdraw manager can be appointed again.

CLGaugeFactory - EMERGENCY_WITHDRAW_ROLE

- Trust level: Fully trusted.
- Permissions: Enables and disables `CLGauge.emergencyWithdraw()` for all gauges of the factory.
- Assumptions: The role only enables emergency withdrawals while gauge settlement on the `LeafVoter` fails. The role never disables emergency withdrawals after they have been enabled.
- Worst-case impact: Keeping emergency withdrawals disabled while settlement fails leaves all staked NFTs of the affected gauges stuck. Enabling them while settlement works lets stakers of every gauge of the factory capture emissions accrued before their deposit, see [Emergency Withdrawals Let New Stakes Capture Past Emissions](#).

LeafVoter - EMERGENCY_COUNCIL_ROLE

- Trust level: Partially trusted.
- Permissions: Sets the emission cap of any registered gauge to zero through `CLGaugeFactory.setEmissionCap`, `setEmissionCaps` or `clearEmissionCap`. `clearEmissionCap` zeroes the cap even if flushing the gauge's fees fails. The role alone cannot set a nonzero cap.
- Worst-case impact: Zeroing every gauge's cap stops all emissions and blocks fee collection, so the staked-liquidity fee share accumulates in each pool's `gaugeFees` until `CAP_ADMIN_ROLE` raises the caps. Cannot affect LP principal or rewards already accrued.

NonfungiblePositionManager - Owner

- Trust level: Partially trusted.
- Permissions: Can replace `tokenDescriptor` (the on-chain NFT metadata/artwork renderer) with any address. Can transfer ownership.

- Worst-case impact: A malicious `tokenDescriptor` causes `tokenURI` calls to revert or return arbitrary data, degrading off-chain display and integrators that rely on on-chain metadata. Does not affect LP principal, fee accrual, or token transfers. The owner has no power over pool accounting or position liquidity.

CLFactory - swapFeeManager (factory-appointed)

- Trust level: Partially trusted.
- Permissions: Controls the default and per-pool swap hook assignments on `CLFactory` via `setDefaultSwapHook` and `setPoolSwapHook`.
- Worst-case impact: Can route all swaps through an arbitrary hook for any pool. A hook setting fees at the ceiling (10%) extracts the most from every swap. A hook setting fees to zero removes LP fee revenue. A reverting hook blocks every swap and flash loan on the pools it is assigned to. Hooks with an `afterSwap` callback write fees unconditionally to the pool's `gaugeFees` slot without checking whether a gauge exists. Assigning a fee-collecting hook to a pool with no gauge permanently locks those fees, since only the gauge can call `collectFees`. The swapFeeManager is trusted not to configure fee-collecting hooks on pools without gauges.

Liquidity Providers / End Users

- Trust level: Untrusted.
- Permissions: Can mint, increase, and decrease positions through `NonfungiblePositionManager`, and migrate positions through `LpMigrator`. Can deposit and withdraw NFTs from `CLGauge` and claim earned and deferred emissions. Can withdraw their own NFTs through `CLGauge.emergencyWithdraw()` while emergency withdrawals are enabled. Referrers can claim their deferred referral emissions.
- Worst-case impact: An LP can only affect their own positions and claimed rewards. They cannot influence other users' accounting, pool state, or protocol configuration. One exception involves `NonfungiblePositionManager.refundETH()`, which is called at the end of `mint`, `increaseLiquidity`, and the swap entry points. It forwards the contract's entire ETH balance to `msg.sender`. If any ETH is left in the contract from a prior payable call (e.g., an incomplete multicall), it is forwarded to the next caller of any of these functions. A contract integrator without a payable fallback will revert on that transfer, bricking those entry points for that integrator until the balance is drained. Any user can trigger this condition by leaving 1 wei in the contract. `LpMigrator` likewise sends its entire token0 and token1 balance to the caller at the end of each migration, so any balance left in it goes to the next caller of a migration involving those tokens.

4.2 External Dependencies

Voter / LeafVoter The gauge system is fully subordinate to the external Voter for its emissions lifecycle. The Voter drives gauge creation (via `gaugeManager`), pushes reward rates via `settleGauge`, mints emissions to LPs and referrers via `mintEmissions`, and accepts forfeited rollover via `forfeitEmissions`. It also manages the `EMERGENCY_COUNCIL_ROLE` that `CLGaugeFactory` checks via `hasRole` to authorize emergency cap-zero operations. The cumulative reward share the Voter reports on `settleGauge` directly determines how much of the reward token is minted to LPs. A Voter that over-reports mints excess tokens, while one that under-reports or stops reporting silently starves gauges of emissions. A Voter whose `settleGauge` reverts blocks gauge deposits, withdrawals, claims and tick-crossing swaps of the pool, since the pool settles the gauge on each of them. A compromised Voter can halt all gauge emissions, drain the protocol's emission budget by inflating the cumulative reward share delta, or grant `EMERGENCY_COUNCIL_ROLE` to any address to zero every gauge's emission cap, freezing fee collection for all staked LPs.

Pool tokens Pool creation is permissionless and does not filter tokens. Pools are assumed to hold standard ERC-20 tokens. A token that reverts or runs code in `transfer`, `transferFrom` or `balanceOf`, or that blocklists an address, only affects pools that contain it. Examples are a token that blocks the Gauge's `VotingRewardsManager`, which makes `collectFees` revert for both tokens of the pool, and a token that re-enters the periphery's ETH helpers while ETH is attached to a call.

5 System Considerations

We leverage this section to highlight findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

SC2 Security Considerations for Pricing a Swap Fee on Trade Size

System Consideration	Version 1
----------------------	-----------

`beforeSwap()` receives `amountSpecified` before the swap loop runs. This is the requested trade amount, and can be more than the amount filled when the pool runs out of liquidity or, more importantly, when the limit price `sqrtpPriceLimitX96` is reached. The `beforeSwap()` hook should not charge a fee based on the specified value.

If large swaps are priced cheaper than small ones: A trader can request a large amount to receive a low fee and combine it with a tight limit price, so that only the intended amount actually trades while paying an artificially low fee.

Similarly, if large swaps are more expensive than small ones: A trader requesting a large trade and accepting a high fee in return could get sandwiched so that the limit price is reached earlier than expected, and end up paying a large trade's fee on a small fill.

A hook whose fee depends on trade size should therefore be implemented in `afterSwap()` rather than in `beforeSwap()`. The filled amount is available there, as `amountSpecified` less `amountSpecifiedRemaining`.

SC3 Security Considerations for the Not-Emitting Restriction

System Consideration	Version 1
----------------------	-----------

`CreationMode.WHEN_NOT_EMITTING` stops an incentive from being created while the pool's gauge is emitting, but `_gaugeEmitting()` returns false whenever the voter reports no gauge, which includes a pool the factory has not deployed yet. A campaign created that early passes the gate and keeps running once the gauge begins emitting.

Pool addresses are deterministic, so campaigns can be prepared for pools that do not exist yet. Only `maxIncentiveStartLeadTime` bounds how far ahead of its start a campaign may be created, so it must be chosen small enough to make that impractical.

SC4 Security Considerations for Gauge Integrators

System Consideration	Version 1
----------------------	-----------

Integrating contracts that stake into a `CLGauge` should consider the following properties:

- The early-unstake penalty can take the whole reward:** `CLGauge` charges a penalty on the emissions of any position claimed or withdrawn while `block.number` is below its `depositBlock` plus `minStakeBlocks`. At the factory defaults of `DEFAULT_MIN_STAKE_BLOCKS = 5` and `DEFAULT_PENALTY_RATE = MAX_PIPS`, the whole balance gets slashed. Nothing in the call reports it, as `claimEmissions()` and `withdraw()` return no amount received.
- The penalty window is evaluated at claim time:** the window uses the configuration live at the time of the claim, not at the time of the deposit. The penalty admin can therefore backrun a deposit, or any position

that has already left the window, by raising `minStakeBlocks` for the gauge or factory-wide, which puts it back inside the window.

3. **Withdrawals defer emissions they cannot mint:** if minting fails, `withdraw()` and `withdrawFrom()` do not revert but add the position's emissions to `deferredEmissions`, so that the position can always be withdrawn. Withdrawals never mint deferred emissions, neither these nor any deferred earlier. Only `claimEmissions()` pays them out, also for an account that no longer has a staked position.
4. **Anyone can add positions to an account's stake set:** `Gauge.depositFor()` credits a stake to an arbitrary `owner`, so any party can grow a chosen account's stake set with dust positions. Enough donated positions make `claimEmissions(address, address)`, `earned(address)` and `stakedValues()`, which operate on the whole set, run out of gas.

An integrating contract must therefore compare each position's `depositBlock` against `CLGaugeFactory.minStakeBlocks()` in the same transaction as the claim, and check `deferredEmissions()` after a withdrawal and call `claimEmissions()` to receive them. It should keep its own record of the positions it staked, and use `claimEmissions(address, address, uint256[])` and `earned(address, uint256[])` with those ids.

In **Version 5**, `CLGauge.emergencyWithdraw()` was added as an escape hatch for staked positions if gauge settlement fails, see [Pools Freeze in Their Current Range if Gauge Settlement Fails](#). However, it only keeps an integration working if the integration:

1. **Does not integrate solely through the Metarouter:** the Metarouter has no emergency withdrawal command.
2. **Withdraws for itself instead of relying on others:** `emergencyWithdraw()` only accepts the stake owner as caller and has no operator variant like `withdrawFrom()`.
3. **Can receive the NFT:** `emergencyWithdraw()` returns it through `safeTransferFrom()`, so a contract must accept it in `onERC721Received()`.
4. **Does not rely on the views not to revert:** `earned()` runs the same settlement and reverts as well.

SC5 `CLGaugeTape` Picks Its Ring by a Fixed 48-Hour Threshold and Does Not Interpolate

System Consideration Version 3

`CLGaugeTape.observe()` returns, for each lookback, the newest stored observation at or before the target timestamp, or the oldest one if the target precedes it.

`CLGaugeTape` has two rings, an hourly ring for hourly observations and a daily ring for daily observations. The ring chosen to locate the observation is given by the lookback period: up to `HOURLY_SPAN` (48 hours, matching the 48 slots of the hourly ring) the hourly ring is chosen, beyond it the daily ring. With hourly writes, the hourly ring covers only about 47 hours plus the age of the newest write, so a lookback just below 48 hours is clamped to the oldest hourly observation, even if the daily ring holds one at or before the target. Conversely, when a gauge settles rarely, a lookback just above 48 hours reads the daily ring even if an hourly observation is closer.

Further, unlike `CLPoolTape.observe()`, the tape does not interpolate. The returned `blockTimestamp` is the time of the write, which can lie far from the target while a gauge does not settle. A comparison of pool and gauge tapes should take this into account and compute the elapsed time from the returned timestamps rather than from the requested lookbacks.

SC6 `try / catch` Does Not Catch Every Failure

System Consideration Version 3

Several call sites in MetaDEX Slipstream rely on `try / catch` to fall back when an external call fails. A `catch` clause only handles a revert inside the callee. Everything the caller does around the call happens in the caller's own frame. If the callee can influence these actions, it can still make the caller revert in a way the `catch` does not handle. The call sites below are compiled with solc 0.7.6 and ABI coder v2.

Failures that escape the catch

1. **Return data that cannot be decoded.** Decoding runs in the caller. It reverts on return data shorter than the declared types and, with ABI coder v2, also reverts on values larger than the declared type: a `uint24` or `uint128` above its maximum.
2. **Address without code.**
 1. **Code check:** the caller checks for code before the call and reverts. Before 0.8.10, including 0.7.6, this applies to every external call. From 0.8.10 on, only to calls without return values.
 2. **Empty return data:** from 0.8.10 on, calls with return values skip the code check, and the empty return data fails decoding as in item 1.
3. **Large dynamic return data.** For dynamic return types, the caller copies the full return data into memory and pays for the memory expansion. This can make the caller run out of gas from memory expansion (returndata bomb).
4. **Typed catch clauses.** `catch Error(string memory)` without a general `catch` lets custom errors, panics and empty reverts through.
5. **Gas exhaustion.** Without an explicit gas limit, the call forwards 63/64 of the remaining gas. A callee that uses all of it leaves the caller 1/64 to finish, which reverts unless the transaction carries roughly 64 times the gas the rest of the execution needs.

Call sites

Callees that are not fully trusted, such as hooks and fee modules, can trigger these failures through a bug or a misconfiguration. Examples:

1. `SwapHookLib.beforeSwap()`, `afterSwap()`, `beforeFlash()`, `getBeforeSwapFee()`, `getAfterSwapFee()`, `getFlashFee()`: swap hook, `uint24`, affected by failures 1, 2.1, 5.
2. `CLFactory.getUnstakedFee()`: unstaked fee module, `uint24`, affected by failures 1, 2.1, 5.

Fully trusted callees

Callees that are fully trusted:

1. `CLGauge._claimOrDeferPositionEmissions()` calling `LeafVoter.mintEmissions()`: no return values, affected by failures 2.1, 5.
2. `CLGaugeFactory` calling `VotingRewardsManager.flushFees()`: no return values, affected by failures 2.1, 5.

The `excessivelySafeCall()` used for the hook and fee module calls in **Version 1** capped the gas, checked the length of the return data and decoded a `uint256`, so it handled failures 1, 2, 3 and 5.

SC7 Security Considerations for Swap Hooks

System Consideration Version 3

In `CLPool`, `swap()` calls the hook's `afterSwap()` after the swap loop has crossed ticks, but before it writes the new `slot0`, `liquidity` and fee growth. During that call, `snapshotCumulativesInside()`, `getFeeGrowthInside()`, `getRewardGrowthInside()` and `CLGauge.earned()` combine the already flipped tick values with the stale `slot0.tick`, for example reporting more seconds inside a range than have elapsed. These are views without the lock, so whoever gets control while the hook runs can read them in that state. State-changing pool and gauge functions are locked and revert.

A swap hook, and every contract it calls such as the pool tape, must therefore never hand control to untrusted code during `afterSwap()`. Integrators must not read these views from within a swap.

SC8 Pools Freeze in Their Current Range if Gauge Settlement Fails

System Consideration Version 5

If `LeafVoter.settleGauge()` reverts for a gauge, every pool action that settles the gauge reverts with it. Staked positions can then only leave through `CLGauge.emergencyWithdraw()`, once `EMERGENCY_WITHDRAW_ROLE` has enabled emergency withdrawals on the `CLGaugeFactory`. For a pool with a gauge, this has the following effects:

1. **Swaps cannot leave the current range:** a swap that crosses an initialized tick settles the gauge first and reverts. Swaps within the current range still succeed.
2. **Only out-of-range positions can be deposited:** `deposit()` settles the gauge only for in-range positions. `withdraw()`, `withdrawFrom()` and `claimEmissions()` settle it for every position and revert. A position deposited out of range can therefore not be withdrawn again until settlement works or emergency withdrawals are enabled. Unstaked positions are unaffected.
3. **Oracles and tapes report a pinned price:** observations are still written, but the tick stays within the current range while the market price moves. The TWAP derived from `tickCumulative` therefore diverges from the market, and the pool tape misses every swap that reverts. Consumers such as dynamic fee modules read stale values.

6 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

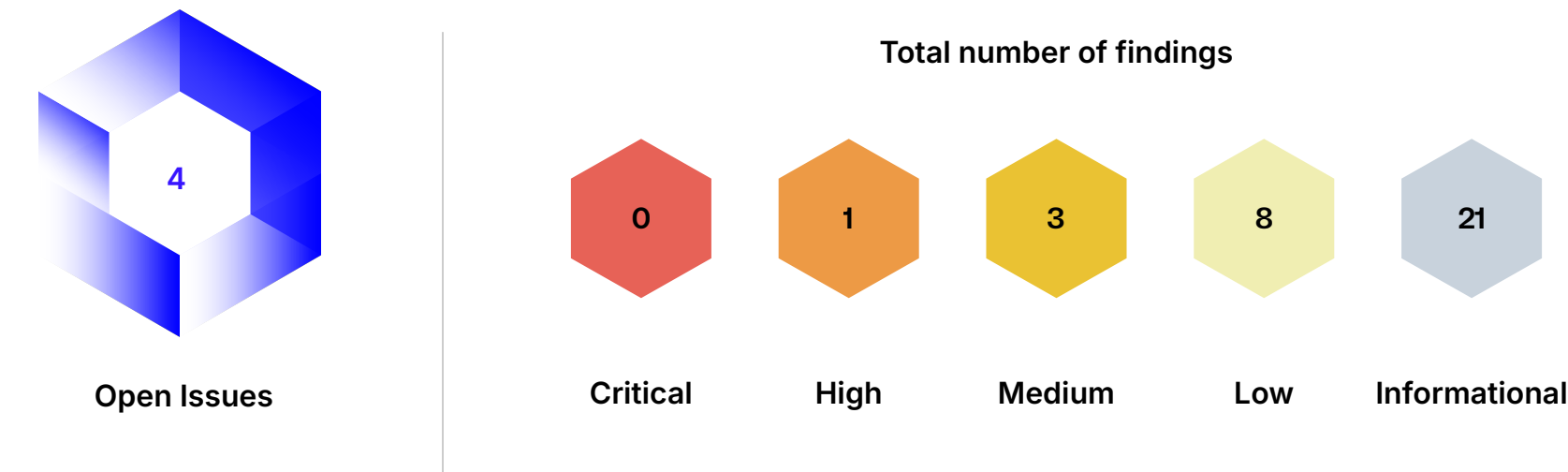
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Non-informational findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

High	#001	setEmissionCap Reverts: LeafVoter Has No emergencyCouncil	Code Corrected	✓
Medium	#002	Chain Suspension Locks Staked NFTs with Pending Rewards	Code Corrected	✓
Medium	#003	Hardcoded Gas Stipends Cannot Adapt to Gas Repricing	Code Corrected	✓
Medium	#004	Griefing via Passing Insufficient Gas	Code Part. Corrected / Risk Accepted	⚠
Low	#005	A Hook Cannot Set a Zero Swap Fee	Acknowledged	⊖
Low	#006	A Zero Refundee Makes Residual Rewards Unrecoverable	Code Corrected	✓
Low	#007	Emergency Council with Cap Admin Rights Can Only Zero Caps	Code Corrected	✓
Low	#008	Ending an Incentive Campaign Can Be Griefed	Code Corrected	✓
Low	#010	Liquidity in Range Past an Incentive's End Still Earns Rewards	Code Corrected	✓
Low	#011	setDefaultCap Retroactively Reprices Unsettled Gauge Accrual	Code Corrected	✓
Low	#030	Token Revert Data Can Be Decoded as a Swap Quote	Risk Accepted	⚠
Low	#033	Emergency Withdrawals Let New Stakes Capture Past Emissions	Risk Accepted	⚠
Info	#009	Griefing via Initial Price	Acknowledged	⊖
Info	#012	A Sequencer Outage Can Lead to Slashing	Code Corrected	✓
Info	#013	Any Contract Is Accepted as the Reward Token of an Incentive	Code Corrected	✓
Info	#014	Claim Amount Is Remapped in Undocumented Ways	Code Corrected	✓
Info	#015	CLPoolTape Called by Swap Hook Instead of Pool Directly	Acknowledged	⊖
Info	#016	Draining a Tick Range Temporarily Locks a Zero-Liquidity NFT	Acknowledged	⊖
Info	#017	Gauge Clone Is Left Uninitialized Across an External Call	Acknowledged	⊖
Info	#018	Incentive Top-Ups Are Emitted as Incentive Creations	Code Corrected	✓

Info	#019	Inconsistent NatSpec on SwapParams	Code Corrected	✓
Info	#020	Interface Mismatch of IGauge	Code Corrected	✓
Info	#021	Magic Value of 0 for Reward Growth	Acknowledged	⊖
Info	#022	Module Unstaked Fees Can Reach 100%, Twice the Factory Cap	Acknowledged	⊖
Info	#023	Package.json Files Entry Omits Contracts/periphery/libraries	Code Corrected	✓
Info	#024	Read-Only Re-Entrancy in NonfungiblePositionManager's Collect	Acknowledged	⊖
Info	#025	Redundant Account Argument in withdrawTokenFrom	Code Corrected	✓
Info	#026	Referral Parameters Are Decoded Without a Length Check	Code Corrected	✓
Info	#027	Reward Is Not Bounded by the Incentive's Unclaimed Balance	Code Corrected	✓
Info	#028	SlipstreamStaker Constructor Misses Zero Address Checks	Code Corrected	✓
Info	#029	withdrawToken Accepts the Zero Address as Recipient	Code Corrected	✓
Info	#031	CLGaugeTape Live Edge Reads Unvalidated Gauges	Acknowledged	⊖
Info	#032	NFTs Sent Directly to a Gauge Cannot Be Recovered	Acknowledged	⊖

7.2 Descriptions

#001 setEmissionCap Reverts: LeafVoter Has No emergencyCouncil

Correctness High Version 1

Code Corrected ✓

`CLGaugeFactory.setEmissionCap` reads the emergency-council address as its first action:

```
address emergencyCouncil = ILeafVoter(leafVoter).emergencyCouncil();
if (msg.sender == emergencyCouncil) { ... return; }
if (hasRole(CAP_ADMIN_ROLE, msg.sender)) { ... return; }
if (hasRole(CAP_OPERATOR_ROLE, msg.sender)) { ... return; }
```

The project's `ILeafVoter` interface declares `emergencyCouncil()` external view returns (address), but the real `LeafVoter` does not implement this function. It manages its emergency council via `EMERGENCY_COUNCIL_ROLE` on `AccessControlEnumerable`, and exposes no `emergencyCouncil()` view. With no matching selector and no fallback on the `LeafVoter`, the external call reverts, and `setEmissionCap` becomes unreachable for all callers — emergency council, `CAP_ADMIN_ROLE`, and `CAP_OPERATOR_ROLE` alike. No gauge cap can be set, changed, or zeroed after deployment.

Code Corrected

`CLGaugeFactory` now checks `EMERGENCY_COUNCIL_ROLE` on the `LeafVoter` instead of calling `emergencyCouncil()`.

#002 Chain Suspension Locks Staked NFTs with Pending Rewards

Security Medium Version 1

Code Corrected ✓

`CLGauge._withdraw()` unconditionally calls `_claimPositions()` as part of every withdrawal:

```
_claimPositions(account, _tokenIds, recipient);
pool.stake(-(uint256(liquidityToStake).toInt256()).toInt128(), tickLower, tickUpper);
...
nft.safeTransferFrom(address(this), msg.sender, tokenId);
```

`_claimPositions` settles the pool and, when a position has accrued LP rewards, calls `voter.mintEmissions()`:

```
if (_amounts.length > 0) {
    voter.mintEmissions(_recipients, _amounts);
}
```

`LeafVoter.mintEmissions` carries a `chainIsActive` modifier that reverts with `ChainNotActive` whenever `chainStatus != Active`. When the chain is suspended, which zeroes `emissionsPerVP` and stops new accrual, any withdrawal attempt on a position with unclaimed LP rewards in `rewardGrowthGlobalX128` fails at this call, blocking the NFT transfer.

`forfeitEmissions` (called for rollover and penalties) carries no such modifier and is unaffected. Only positions with a non-zero LP reward component are locked. Positions that accrued nothing, or whose rewards were fully claimed before suspension, can still be withdrawn.

There is no withdrawal path that bypasses `_claimPositions`, and `claimEmissions` routes through the same function, so the accrued rewards cannot be drained first. The lock persists for the duration of the suspension.

Because the `Suspended` → `Active` transition is the only exit from `Suspended`, the chain can always be resumed. However, governance may delay that resumption arbitrarily, extending the lock.

Code Corrected

Withdrawals now mint emissions in a `try / catch` and defer them if minting fails. Deferred emissions are paid out with the next `claimEmissions()`.

#003 Hardcoded Gas Stipends Cannot Adapt to Gas Repricing

Security

Medium

Version 1

Code Corrected ✓

Every gas-limited external call in scope sizes the callee's budget with a compile-time constant. None of these values has a setter, and the constants section holding the hook values carries a `// TODO(future PRs/commits): review and assess sanity of these values` note. The affected sites are:

1. `SwapHookLib._safeCall()` (`_SAFE_CALL_GAS_LIMIT`, `500_000`): a `private constant` forwarded to the swap hook on all three hook entry points, `beforeSwap()`, `afterSwap()` and `beforeFlash()`.
2. `CLFactory.getSwapFee()` (`200_000`): a literal forwarded to `swapFeeModule`.
3. `CLFactory.getUnstakedFee()` (`200_000`): a literal forwarded to `unstakedFeeModule`.
4. `CLFactory.createPool()` (`gasleft()`): forwards all remaining gas, but fixes the work the callee has to perform through `DEFAULT_POOL_TAPE_CARDINALITY`, an `internal constant` set to `100`.

```
(bool _success, bytes memory _data) =  
    _swapHook.excessivelySafeCall(_SAFE_CALL_GAS_LIMIT, _SAFE_CALL_RETURN_DATA_SIZE,  
    _hookData);
```

In each of these instances an out-of-gas failure in the callee is swallowed silently rather than reverting, so a stipend that becomes too small degrades behavior without any on-chain signal. `SwapHookLib` is inlined into the pool implementation, so its limit is fixed for every pool cloned from that implementation and can only be changed by deploying a new implementation. Existing pools keep the old value.

The Glamsterdam upgrade raises the cost of zero-to-nonzero storage writes, so stipends sized against today's costs can fall short afterward. The cost of a zero-to-nonzero storage write is expected to increase from 20,000 to 100,000 gas, which turns a `200_000` stipend from ten such writes into two.

Note that Glamsterdam is also expected to add a separate `state_gas_reservoir` for storage writes, populated by setting a gas limit above the per-transaction cap of roughly 16 million gas, so the affected calls would not be permanently bricked. They would, however, require the caller to set a very large initial gas limit to succeed, with potentially problematic UX consequences, and insufficient gas griefing attacks would become trivially achievable.

The pre-warm in `createPool()` is exposed the same way from the other side. Its cardinality target of `100` fixes the number of observation slots the pool tape must write in one call, and because the value is a constant it cannot be lowered to fit a repriced cost. The pool tape implementation is not part of this scope, so the actual cost of that call cannot be bounded here.

More generally, because the stipends are hardcoded, the protocol cannot raise them in response to any later repricing of state costs, which is always possible and could lead to more critical flaws.

Code Corrected

The hardcoded stipends were removed. The fee module lookup in `CLFactory` now uses `try / catch` passing 63/64 of the available gas, and in **Version 3** the hook entry points in `SwapHookLib` call the hook directly.

#004 Griefing via Passing Insufficient Gas

Security

Medium

Version 1

Code Partially Corrected / Risk Accepted 

In `SwapHookLib` and `CLFactory`, several external calls are wrapped in `excessivelySafeCall` or `excessivelySafeStaticCall` and the result is handled by branching on success. The gas argument is only an upper bound on what reaches the callee. Per EIP-150, the callee can use at most `min(requested, gasleft() - gasleft() / 64)`, and the EVM does not revert when less than the requested amount is available. These wrappers also cannot tell a genuine revert from an out-of-gas, as the EVM does not expose the difference.

```
(bool success, bytes memory data) =
    swapFeeModule.excessivelySafeStaticCall(200_000, 32,
        abi.encodeWithSelector(IFeeModule.getFee.selector, pool));
if (success) { /* use module fee */ } // else fall back to the default fee
```

The gas remaining at the call site depends on the top-level transaction gas limit, which the sender of the transaction chooses. A malicious caller can pick a limit such that `gasleft()` at the call site is below the requested stipend. The callee then runs out of gas, while the one sixty-fourth retained by the calling frame is still enough to finish the outer logic. The caller is left with a free choice between the callee's result and the fallback, and takes whichever is more favorable. There are multiple sites where this is possible:

1. `SwapHookLib.beforeSwap() ( 500_000 )`: falls back from the hook's fee to `tickSpacingToFee[tickSpacing]`, which then prices the entire swap.
2. `SwapHookLib.beforeFlash() ( 500_000 )`: falls back from the hook's flash fee to `tickSpacingToFee[tickSpacing]`.
3. `SwapHookLib.afterSwap() ( 500_000 )`: drops the post-swap fee that would have been credited to `gaugeFees`.
4. `CLFactory.getUnstakedFee() ( 200_000 )`: falls back to `defaultUnstakedFee`. `CLPool` reads this through `unstakedFee()` on every fee-accruing step of a swap or flash, so the fallback shifts the fee split between staked and unstaked liquidity.
5. `CLFactory.getSwapFee() ( 200_000 )`: falls back to `tickSpacingToFee[tickSpacing]` and is exposed to integrators through `CLPool.fee()`. Within the module, `DynamicSwapFeeModule._getDynamicFee()` additionally catches a failing `observe()` and returns a zero surcharge, so starving that inner call lowers the reported fee without failing the outer read.
6. `CLFactory.createPool() ( gasleft() )`: an out-of-gas in the pool tape's `increaseObservationCardinalityNext()` silently drops the initial observation-cardinality bump.

In practice, whether this is exploitable depends heavily on the callee's actual gas use and on the gas cost of the execution thereafter, which still has to succeed. Both are notoriously hard to calculate, though. Where it is feasible, the caller chooses the fee that prices their own swap or flash loan, the split between staked and unstaked liquidity, and whether the gauge is credited a post-swap fee at all, with nothing on-chain flagging that the callee never ran.

Changes in **Version 2**

The gas caps were removed. The calls into the swap hook and into the unstaked fee module now use `try / catch` without a gas limit, passing 63/64 of the available gas.

Changes in Version 3

Three more sites were added:

7. `CLGauge` withdrawals: defer the position's emissions when the mint fails.
8. `CLGaugeFactory.clearEmissionCap()`: clears the cap even when flushing the gauge's fees fails.
9. `CLGaugeFactory.recordCLGaugeTape()`: drops the gauge tape record.

Code Partially Corrected

Sites 5 and 6 no longer exist in **Version 3**. `CLFactory` no longer queries a swap fee module, and `createPool()` no longer calls the pool tape.

Sites 1 to 3 no longer catch failures in **Version 3**. `SwapHookLib` calls the hook directly, so a failing hook reverts the swap or flash.

Site 4 remains exposed. It no longer passes a gas limit, but the `try/catch` still cannot distinguish a revert from an out-of-gas failure. Forcing the fallback now requires the fee module to run out of 63/64 of the remaining gas while the one sixty-fourth retained by the calling frame still completes the transaction. This is only possible if the module consumes more than 63 times the gas of the execution that follows the call. The rest of the swap, its state writes, the token transfers and the swap callback still follow, so this requires a fee module with a very high gas use. What a caught out-of-gas failure leads to:

4. `CustomUnstakedFeeModule.getFee()` reads a single stored value and has no revert path. The gauge, activation and emission cap reads in `getUnstakedFee()` just before it are not wrapped. A caught out-of-gas failure applies `defaultUnstakedFee` to the swap or flash.

With the current fee module implementations, site 4 has no reachable reverts to handle, only an out-of-gas failure may reach its `catch`.

Sites 7 to 9 each handle a legitimate failure:

7. The mint fails while the chain is suspended, which the gauge could also check directly by querying `chainStatus()` on the `LeafVoter`. A caught out-of-gas failure only defers the position's emissions to `deferredEmissions`, which are paid out on the account's next claim.
8. Flushing fails when collecting the fees from the gauge fails. Only the emergency council can call `clearEmissionCap()`, so only it can cause the out-of-gas failure, which leaves the gauge's fees in the pool until a cap is set again.
9. The relay keeps a failing gauge tape from blocking gauge settlement, and reverts if `gasleft()` does not exceed the configured limit. The check does not account for the one sixty-fourth retained by the calling frame, nor for the gas spent between the check and the call. When `gasleft()` is slightly above the limit, the tape still receives less than it, so the caller of a user action that settles the gauge can drop the tape record within this window. The NatSpec of `recordCLGaugeTape()` acknowledges this.

Risk Accepted

Dromos Labs accepts the remaining risk regarding the potential out-of-gas griefing attacks.

#005 A Hook Cannot Set a Zero Swap Fee

Design

Low

Version 1

Acknowledged 

In `SwapHookLib`, `beforeSwap()`, `afterSwap()` and `beforeFlash()` all reach the hook through `_safeCall()`. That helper returns two values, a flag saying whether a usable fee was obtained and the fee itself.

```
(bool _success, bytes memory _data) =
    _swapHook.excessivelySafeCall(_SAFE_CALL_GAS_LIMIT, _SAFE_CALL_RETURN_DATA_SIZE,
        _hookData);

if (_success && _data.length == _SAFE_CALL_RETURN_DATA_SIZE) {
    uint256 _feeRaw = abi.decode(_data, (uint256));
    if (_feeRaw <= _SWAP_FEE_CEIL) {
        _fee = uint24(_feeRaw);
        _okFee = _success;
    }
}
```

The flag is the only thing that separates a failed call from a successful one. There are three scenarios:

1. **Hook returns a valid non-zero fee:** flag true, fee above zero.
2. **Hook fails, returns malformed data, or returns a fee above `_SWAP_FEE_CEIL`:** flag false, fee zero.
3. **Hook returns a valid fee of zero:** flag true, fee zero.

In the third scenario, only `beforeFlash()` would use the flag. `beforeSwap()` checks that the fee is above zero and would then treat the hook as having failed, so it replaces the hook's zero fee with the factory tick-spacing fee. `afterSwap()` likewise ignores the flag and acts only when the fee is above zero, though there skipping the branch and applying a zero fee have the same effect.

Having each hook resolve the same helper's result by its own rule makes future changes to the hook interface more complicated, as one has to reason about every entry point separately.

Acknowledged

Dromos Labs is aware of this behavior and states:

Regarding inability to return 0 fee on swap methods, this is expected and acceptable.

#006 A Zero Refundee Makes Residual Rewards Unrecoverable

Design

Low

Version 1

Code Corrected 

`createIncentive` validates the reward amount, timing, and duration but does not check that `key.refundee` is a non-zero address. `endIncentive` sends the unclaimed balance directly to `key.refundee`:

```
TransferHelperExtended.safeTransfer(address(key.rewardToken), key.refundee, refund);
```

Because the incentive ID is the hash of the full key, an incentive created with `refundee == address(0)` cannot be superseded by one with a valid refundee. After `endTime + REFUND_DELAY`, any tokens remaining in `totalRewardUnclaimed + totalRewardSlashed` are sent to the zero address and permanently lost.

Code Corrected

`SlipstreamStaker` and its libraries were removed in **Version 3**.

#007 Emergency Council with Cap Admin Rights Can Only Zero Caps

Design

Low

Version 1

Code Corrected ✓

`setEmissionCap` checks the emergency-council identity before the `CAP_ADMIN_ROLE` grant and returns early from the emergency branch:

```
address emergencyCouncil = ILeafVoter(leafVoter).emergencyCouncil();
if (msg.sender == emergencyCouncil) {
    require(_cap == 0, 'NA');
    ...
    return; // CAP_ADMIN branch is never reached
}
if (hasRole(CAP_ADMIN_ROLE, msg.sender)) { ... }
```

An address that simultaneously holds `emergencyCouncil` status and `CAP_ADMIN_ROLE` is always routed to the emergency branch, which requires `_cap == 0`. It cannot use its `CAP_ADMIN_ROLE` powers to set non-zero caps. The condition is recoverable by granting `CAP_ADMIN_ROLE` to a separate address, but the ordering makes the dual-role configuration silently restrictive.

Code Corrected

The emergency council branch of `setEmissionCap()` is now only taken for a zero cap.

#008 Ending an Incentive Campaign Can Be Griefed

Design

Low

Version 1

Code Corrected ✓

`SlipstreamStaker.endIncentive()` refunds the refundee only once the incentive's `numberOfStakes` has been reduced to zero, so a griever can stake a large number of dust positions on an incentive campaign and refuse to unstake. Clearing them without the stakers is possible, since `unstakeToken()` becomes unpermissioned once `endTime + REFUND_DELAY` passes, but it costs the refundee one call per position.

The griever moreover saves gas. `unstakeToken()` performs the pool snapshot and the reward computation, and credits `reward - slash` to the position's deposit owner, so whoever clears the incentive pays for that settlement while the staker is left with a credited balance in the `rewards` mapping to collect. Never unstaking therefore imposes a larger cost on the incentive creator than on the griever.

Code Corrected

`SlipstreamStaker` and its libraries were removed in `Version 3`.

#010 Liquidity in Range Past an Incentive's End Still Earns Rewards

Correctness

Low

Version 1

Code Corrected ✓

`SlipstreamStaker.unstakeToken()` passes the pool's live `secondsPerLiquidityInsideX128` and `block.timestamp` to `RewardMath.computeRewardAmount()` without clamping either to the incentive's `endTime`:

```
secondsInsideX128 = (secondsPerLiquidityInsideX128 - secondsPerLiquidityInsideInitialX128)
    * liquidity;
uint256 totalSecondsUnclaimedX128 = ((Math.max(endTime, currentTime) - startTime) << 128) -
    totalSecondsClaimedX128;
reward = FullMath.mulDiv(totalRewardUnclaimed, secondsInsideX128,
    totalSecondsUnclaimedX128);
```

`_stakeToken()` accepts new stakes until `endTime`, after which the pool oracle continues accumulating `secondsPerLiquidityInside` for any position still in range. Because the denominator grows with `Math.max(endTime, currentTime)` to match, the per-second rate is unchanged. A position staked just before `endTime` and held in range afterward is paid for time spent after the campaign closed.

The magnitude scales with how long the position is held past `endTime`. A position opened in the last block and left in range through the one-hour `REFUND_DELAY` window earns for that extra hour. On a one-week incentive, that is about 0.6% of whatever reward is still unclaimed, though the fraction grows for shorter campaigns.

At the extreme, a staker who enters one second before `endTime` and keeps the position in range for an additional duration equal to the full campaign length `D` accumulates `D + 1` seconds inside while the denominator grows to `2D`, capturing roughly 50% of the unclaimed pot. In both cases the cost is borne by the refundee (reduced residual) and any stakers who remain.

Code Corrected

`SlipstreamStaker` and its libraries were removed in **Version 3**.

#011 setDefaultCap Retroactively Reprices Unsettled Gauge Accrual

Correctness Low Version 1

Code Corrected ✓

`setEmissionCap` checkpoints a gauge before modifying its cap by calling `ILeafVoter(leafVoter).settleGauge(_gauge). setDefaultCap` changes the effective cap of every gauge whose stored entry is `DEFAULT_CAP_INDICATOR` without any checkpoint:

```
function setDefaultCap(uint128 _cap) external override {
    require(hasRole(CAP_ADMIN_ROLE, msg.sender), 'NA');
    require(_cap != 0, 'ZC');
    defaultCap = _cap;
    emit DefaultCapSet({_cap: _cap});
}
```

`emissionCap(gauge)` resolves the cap at settlement time via `cap == DEFAULT_CAP_INDICATOR ? defaultCap : cap`. Because settlement is deferred until the next interaction with each gauge, a `setDefaultCap` call changes the cap that will be applied retroactively to the entire unsettled period for every affected gauge, rather than taking effect only from the next settlement forward.

Code Corrected

`createGauge()` now stores the current `defaultCap` as the gauge's cap, so `setDefaultCap()` only affects gauges created afterward.

#030 Token Revert Data Can Be Decoded as a Swap Quote

Security

Low

Version 3

Risk Accepted 

`CLPool.balance0()` and `balance1()` read the pool's token balance with a typed `balanceOf` call, which bubbles up the token's revert data.

`swap()` reads the input-side balance before it calls `uniswapV3SwapCallback`. A token whose `balanceOf` reverts with exactly 96 bytes therefore makes `swap()` revert with a payload of the attacker's choosing. The quoters (`Metaquoter`, `QuoterV2`, `MixedRouteQuoterV1 / V2 / V3`) treat any 96-byte revert from `swap()` as the encoded `(amount, sqrtPriceX96After, tickAfter)` and never run their callback.

`CLFactory.createPool` is permissionless and accepts any token pair, and the resulting pool is registered with the `FactoryRegistry`. It passes the quoters' pool authentication. The quote for such a pool is attacker-controlled. In a multi-hop quote the forged amount also becomes the input of the next hop.

Only quotes of pools that contain a token controlled by the attacker are affected. Swaps are not affected.

Risk Accepted

Dromos Labs has acknowledged the risk and states:

We decided to acknowledge this because it'd likely be malicious tokens triggering this, and it affects quoters and off-chain calls which are lower stakes.

#033 Emergency Withdrawals Let New Stakes Capture Past Emissions

Trust

Low

Version 5

Risk Accepted 

`EMERGENCY_WITHDRAW_ROLE` is meant to open an escape hatch for gauges whose settlement fails. Enabling emergency withdrawals, however, also affects every gauge of the `CLGaugeFactory` that still settles. When an in-range position is emergency-withdrawn, `CLGauge.settleGauge()` returns zero without calling the `LeafVoter`, but `CLPool` still sets `lastUpdated` to the current timestamp. The emissions accrued since the gauge's last settlement therefore stay unpulled, while the pool skips settlement for the rest of the block.

A staker can therefore emergency-withdraw a dust in-range position staked earlier, and then deposit a large in-range position in the same transaction. The deposit does not settle the gauge, so the next settlement spreads all emissions accrued before the deposit over the staked liquidity, including the new position. A swap in the same block that moves the price into another range does not settle the gauge either, so the next settlement credits these emissions to the stakers of that range instead.

The role can therefore expose the stakers of every gauge of the factory to losing unsettled emissions to new stakes. Even a legitimate activation for a few failing gauges does so on all other gauges of the factory.

Risk Accepted

Dromos Labs has accepted the risk of allowing this behavior.

#009 Griefing via Initial Price

Informational Version 1

Acknowledged 

`CLFactory.createPool()` is unpermissioned and puts no bound on `sqrtPriceX96` before passing it to `CLPool.initialize()`. The only check is the one inside `TickMath.getTickAtSqrtRatio()`, so a caller can pick any price `sqrt(1.0001^tick)` with the tick anywhere in `[-887,272, 887,271]`. The upper bound is one below `MAX_TICK` because the price can never reach `MAX_SQRT_RATIO`.

```
require(tickSpacingToFee[tickSpacing] != 0);
require(_getPool[token0][token1][tickSpacing] == address(0));
pool =
  Clones.cloneDeterministic({master: poolImplementation, salt: keccak256(abi.encode(token0,
    token1, tickSpacing))});
CLPool(pool)
  .initialize({
    // ...
    _sqrtPriceX96: sqrtPriceX96
  });
```

Only one pool can exist for each combination of `token0`, `token1` and `tickSpacing`. Once someone initializes a pool at a skewed price, no replacement can be made anymore, and integrators that compute the pool address are forced to use the pool with the skewed initial price. The enabled tiers are 1, 50, 100, 200 and 2000, so an attacker could create pools for every popular token pair in every tier ahead of time.

The price can only be moved by `swap()`, so fixing the price requires swapping across the whole distance. Each loop iteration in `swap()` covers one full tick-bitmap word or `256 * tickSpacing` ticks if no tick in the word is initialized in `TickBitmap.nextInitializedTickWithinOneWord()`.

Skipping an empty bitmap word is a single cold `SLOAD`, so 2,100 gas. The number of words that need to be iterated through is `wordPos = (tick / tickSpacing) >> 8` over the distance walked.

Take a concrete case. The attacker creates the pool at the minimum price while the honest price sits close to 1 (i.e. near tick 0), which is where a stablecoin pair trades. Fixing the price then means walking from `MIN_TICK` to tick 0, a little over half the tick range.

No position has ever been minted, so every bitmap word on the way is empty. The walk touches `1 + ceil(-MIN_TICK / (256 * tickSpacing))` words, where the ceiling covers the partial word at the `MIN_TICK` end and the `1` is the word holding tick 0:

tickSpacing	storage operations = $887,272 / (256 * s)$	rough gas
1	~3,467 cold <code>SLOAD</code>	~13M
50	~71 cold <code>SLOAD</code>	~250k
100	~36 cold <code>SLOAD</code>	~126k
200	~19 cold <code>SLOAD</code>	~67k
2000	~3 cold <code>SLOAD</code>	~11k

Crossing an **initialized tick** is far more expensive. The worst case is a tick never crossed on a pool that has accrued growth. Here crossing could cost at least:

```

6 * 2,100 cold SLOAD, all six slots of Tick.Info
+ 3 * 20,000 SSTORE 0 -> nonzero (feeGrowthOutside0X128,
                                feeGrowthOutside1X128,
                                rewardGrowthOutsideX128)
+ 1 * 2,900 SSTORE nonzero -> nonzero (secondsPerLiquidityOutsideX128)
+ 2 * 100 SSTORE repeat (tickCumulativeOutside, secondsOutside)
= 75,700 gas per crossed tick

```

Glamsterdam makes this worse. EIP-8037 raises the cost of creating new state, taking a fresh storage slot from 20,000 to an estimated 97,920 gas, so the worst-case crossing moves from 75,700 to roughly 310,000.

Integrators must therefore be careful before reading data from or calling into any pool, and check that it was initialized at a sound price or was at least moved to a good state later. Neither the observation history nor the spot price is sufficient on its own, since `OracleLibrary.getOldestObservationSecondsAgo()` can report a long history while `consult()` can average observations that are largely meaningless.

Acknowledged

Dromos Labs has acknowledged the issue, but has decided to keep the code unchanged. They provided the following reasoning:

This issue is also present on Uniswap V3. I'm not quite sure if there's any sane mitigation that would allow to keep pool creation permissionless. And there's no external reference price that would allow checking if the passed `sqrtPriceX96` makes actual sense, or if it is inflated.

#012 A Sequencer Outage Can Lead to Slashing

Informational Version 1

Code Corrected ✓

In `SlipstreamStaker`, the early-unstake penalty window is counted in blocks. `unstakeToken()` becomes unpermissioned `REFUND_DELAY` after `endTime`, which is one hour. The constructor reconciles the two by assuming a minimum rate of block production, `microsPerBlock`. A sequencer outage could lead to fewer than `minStakeBlocks` blocks produced, so a position staked shortly before `endTime` could get slashed.

Code Corrected

`SlipstreamStaker` and its libraries were removed in `Version 3`.

#013 Any Contract Is Accepted as the Reward Token of an Incentive

Informational Version 1

Code Corrected ✓

`SlipstreamStaker.createIncentive()` does not validate `key.rewardToken`. It pulls the reward through `TransferHelperExtended.safeTransferFrom()`, which only requires that the address is a contract and then calls `transferFrom()` on it. A call that does not revert and returns empty data is treated as a success.

```

(bool success, bytes memory data) =
    token.call(abi.encodeWithSelector(IERC20.transferFrom.selector, from, to, value));
require(success && (data.length == 0 || abi.decode(data, (bool))), 'STF');

```

Passing `nonfungiblePositionManager` as `key.rewardToken` therefore moves the caller's position with token id `reward` into the staker, since the ERC-721 `transferFrom()` shares the ERC-20 selector and returns

no data. It only requires the staker to be approved for that token id. As this is a plain transfer and not a safe transfer, `onERC721Received()` is not invoked and no `deposits[tokenId]` entry is written.

The token id is accounted as a reward amount in `totalRewardUnclaimed`, and `unstakeToken()` consumes it and credits fractions of it to `rewards[key.rewardToken][deposit.owner]`. These fractions can in theory match existing NFT IDs that are currently staked in the contract. However, the IDs cannot be paid out.

`endIncentive()` and `claimReward()` transfer tokens with `TransferHelperExtended.safeTransfer()`:

```
(bool success, bytes memory data) = token.call(abi.encodeWithSelector(IERC20.transfer.selector, to, value));
require(success && (data.length == 0 || abi.decode(data, (bool))), 'ST');
```

The position manager implements no `transfer(address,uint256)` and has no fallback function, so both calls revert and the position remains locked in the staker.

The impact is confined to the incentive creator, who has to move their own position and approve the staker for it. Beyond that, NFT token ids enter the reward accounting as amounts. The resulting `totalRewardUnclaimed`, `totalRewardSlashed`, and per-user `rewards` balances have no relation to a transferable reward balance, and the resulting state is not one we can fully reason about.

Code Corrected

`SlipstreamStaker` and its libraries were removed in **Version 3**.

#014 Claim Amount Is Remapped in Undocumented Ways

Informational Version 1

Code Corrected ✓

In `SlipstreamStaker`, `_claimReward()` treats `amountRequested` as a sentinel rather than as the amount to transfer. Zero means claim the whole balance, and any value above the account's `rewards` balance is silently reduced to that balance.

```
reward = rewards[rewardToken][account];
if (amountRequested != 0 && amountRequested < reward) {
    reward = amountRequested;
}
```

Neither behavior appears in the documentation. `ISlipstreamStaker` documents `claimReward()` as transferring "`amountRequested` of accrued `rewardToken` rewards from the contract to the recipient `to`".

Code Corrected

`SlipstreamStaker` and its libraries were removed in **Version 3**.

#015 CLPoolTape Called by Swap Hook Instead of Pool Directly

Informational Version 1

Acknowledged ⊖

The `afterSwap` hook invoked by the `CLPool` calls `CLPoolTape.record()` with data (fees, volumes, tick, staked liquidity) that the pool already has at the end of its swap execution.

```
bytes memory _calldata = abi.encodeWithSelector(
    ICLPoolTape.record.selector,
```

```

_pool,
ICLPoolTape.CLPoolTapeData({
  /// @dev Fee is always paid on the input amount.
  fee0: _swapParams.zeroForOne ? _fee : 0,
  fee1: _swapParams.zeroForOne ? 0 : _fee,
  volume0: _swapParams.zeroForOne ? uint128(_inputAmount) : uint128(_outputAmount),
  volume1: _swapParams.zeroForOne ? uint128(_outputAmount) : uint128(_inputAmount),
  // ...
  tick: _swapParams.tick,
  stakedLiquidity: _afterSwapParams.stakedLiquidity
})
);

```

The hook receives this data only because it was forwarded via `AfterSwapParams`, so it is acting as a pass-through with no insight of its own. Moreover, the pool tape itself is not configurable per pool, so the values stored should be identical regardless of the specific swap hook used. Currently, every new swap hook would have to include the same call to the pool tape to record the parameters provided by the pool.

Acknowledged

Dromos Labs has decided to keep the code unchanged.

#016 Draining a Tick Range Temporarily Locks a Zero-Liquidity NFT

Informational Version 1

Acknowledged 

`CLGauge._withdraw()` calls `nft.collect()` before unstaking, which triggers `pool.burn(tickLower, tickUpper, 0)` on the NFT manager's aggregate pool position for the NFT's tick range. `Position.update()` requires `liquidity > 0` when the delta is zero:

```

if (liquidityDelta == 0) {
  require(_self.liquidity > 0, 'NP'); // disallow pokes for 0 liquidity positions
}

```

A zero-liquidity NFT contributes nothing to this aggregate. If every other NFT that shares the same `(tickLower, tickUpper)` pair has been burned or had its liquidity removed, the NFT manager's aggregate position reaches zero, and the `collect()` call in `_withdraw()` reverts with `'NP'`, blocking withdrawal. The condition resolves as soon as any other position with the same tick range adds liquidity back.

Acknowledged

Dromos Labs is aware of the potential issue, but has decided to keep the code unchanged.

#017 Gauge Clone Is Left Uninitialized Across an External Call

Informational Version 1

Acknowledged 

In `CLGaugeFactory`, `createGauge()` deploys the gauge with `Clones.cloneDeterministic()` and records the factory state for it, then calls `createRewards()` on the voting rewards factory, and only afterwards runs `_initializeGauge()`. The gauge is therefore deployed and externally reachable in an uninitialized state for the duration of that external call. Nothing can act on it in that window, since `CLGauge.initialize()` admits only the gauge factory and only while `pool` is still the zero address, and every other entry point reads through `pool`. The `votingRewardsFactory` is also an immutable set at construction and checked against the zero address there.

Acknowledged

Dromos Labs states:

The trusted voting rewards factory must not transfer control to untrusted external code during this call.

#018 Incentive Top-Ups Are Emitted as Incentive Creations

Informational Version 1

Code Corrected ✓

Calling `SlipstreamStaker.createIncentive()` with the same incentive key a second time tops up the `totalRewardUnclaimed` of the existing incentive, as long as it is called at or before `key.startTime`. Since `_stakeToken()` admits stakes from `key.startTime` onwards, a campaign created with `key.startTime = block.timestamp` can be topped up while positions are already staked.

The function nevertheless emits `IncentiveCreated`, with `reward` set to the top-up amount instead of the incentive's total reward. This can mislead integrators, and it violates the specification, which documents `createIncentive()` as creating a new incentive program and `reward` as the amount of reward tokens to be distributed.

Code Corrected

`SlipstreamStaker` and its libraries were removed in `Version 3`.

#019 Inconsistent NatSpec on SwapParams

Informational Version 1

Code Corrected ✓

`ISwapHook.SwapParams` documents `sqrtPriceX96` and `tick` as the pool's pre-swap price and tick. That holds for `beforeSwap()`, which `CLPool.swap()` fills from the cached `slot0` values, but not for `afterSwap()`, which it fills from the swap loop's final state and whose own parameter documentation describes the same struct as the final pool state. Since one struct serves both entry points, the documentation is inconsistent.

Code Corrected

The NatSpec on `SwapParams` was updated. `sqrtPriceX96` and `tick` are now documented as pre-swap values in `beforeSwap()` and post-swap values in `afterSwap()`.

#020 Interface Mismatch of IGauge

Informational Version 1

Code Corrected ✓

`IGauge` in this repository is not consistent with the interface of the same name in the metadex repository. The MetaDEX Slipstream declaration:

- omits the `voter()` getter
- adds a `ClaimFees` event
- types `gaugeFactory()` as `ICLGaugeFactory`, not `address`
- replaces the custom errors with `require` strings, since Solidity 0.7.6 does not support them

An integration built against the metadex interface therefore cannot decode this gauge's reverts and cannot query the `voter()` function on it.

Code Corrected

The differences between the two `IGauge` interfaces have been resolved.

As Solidity 0.7.6 does not support custom errors, the gauge still reverts with `require` strings instead of the metadex errors.

#021 Magic Value of 0 for Reward Growth

Informational Version 1

Acknowledged 

In `CLGauge._deposit()`, the call `pool.getRewardGrowthInside(tickLower, tickUpper, 0)` passes a literal `0` to signal that the current `rewardGrowthGlobalX128` should be read from state. A named constant would make the intent of that argument more explicit.

Acknowledged

Dromos Labs is aware of the issue but decided to keep the code unchanged.

#022 Module Unstaked Fees Can Reach 100%, Twice the Factory Cap

Informational Version 1

Acknowledged 

`CLFactory.setDefaultUnstakedFee` enforces `_fee <= 500_000` (50%).
`CustomUnstakedFeeModule.setCustomFee` enforces the same `MAX_FEE = 500_000`. However, the acceptance guard in `getUnstakedFee` is twice as permissive:

```
try IFeeModule(unstakedFeeModule).getFee(pool) returns (uint24 fee) {
    if (fee <= 1_000_000) return fee;
} catch {}
```

A buggy or malicious fee module can return any value up to `1_000_000` (100%), and the factory will forward it to the pool without clamping. At a 100% unstaked fee, every swap token owed to unstaked LPs is redirected to staked LPs, leaving unstaked liquidity providers with zero fee income.

Acknowledged

Dromos Labs has acknowledged the risk but has decided to keep the code unchanged.

#023 Package.json Files Entry Omits Contracts/periphery/libraries

Informational Version 1

Code Corrected 

The root `package.json` `files` array contains the entry `"contracts/peripherylibraries"` (missing the path separator) instead of `"contracts/periphery/libraries"`:

```
"files": [
  "contracts/periphery/base",
  "contracts/periphery/interfaces",
  "contracts/peripherylibraries",
  ...
]
```

npm matches paths literally, so `contracts/periphery/libraries/` is not included in the published package. Consumers who install the package and import from paths that are in scope (`contracts/periphery/interfaces`, `contracts/periphery/base`) but which transitively depend on `contracts/periphery/libraries` will receive broken imports.

Code Corrected

The typo was resolved.

#024 Read-Only Re-Entrancy in NonfungiblePositionManager's Collect

Informational Version 1

Acknowledged 

`NonfungiblePositionManager.collect()` reduces `tokensOwed0` and `tokensOwed1` only after `CLPool.collect()` has already transferred the tokens out. If the token hands control to the recipient during that transfer, the view function `positions()` still reports amounts that have already been paid out. Re-entrant writes that could exploit these stale values have to pass back through `CLPool`, where the `lock` modifier rejects them. However, integrations consuming the views are not covered by that guard and could over-value a position.

Acknowledged

Dromos Labs responded:

Note that the same behavior exists in prior versions of the code.

#025 Redundant Account Argument in withdrawTokenFrom

Informational Version 1

Code Corrected 

In `SlipstreamStaker`, `withdrawTokenFrom()` takes an `account` argument and then requires it to equal `deposits[tokenId].owner`, a value it has already loaded into memory. The argument can therefore hold nothing else. The only difference it makes is the non-zero check on it, which, however, is equivalent to requiring the deposit to be initialized as enforced by the code. The argument is therefore redundant.

Code Corrected

`SlipstreamStaker` and its libraries were removed in `Version 3`.

#026 Referral Parameters Are Decoded Without a Length Check

Informational Version 1

Code Corrected ✓

In `CLGaugeFactory`, `_applyReferralParams()` accepts the factory data whenever it is non-empty and its first byte is `PARAMS_REFERRAL`, and `_decodeReferral()` then decodes the remainder with `abi.decode(_params[1:], (address, uint256))` without checking how long it is. If data is longer than the one flag byte plus two words, the tail is ignored, while data shorter than that reverts inside the decode. The equivalent check in the V2 `GaugeFactory` checks that the size is exactly `_REFERRAL_PARAMS_LENGTH` (i.e. 65).

Code Corrected

`_decodeReferral()` now requires the referral parameters to be exactly 65 bytes long.

#027 Reward Is Not Bounded by the Incentive's Unclaimed Balance

Informational Version 1

Code Corrected ✓

`SlipstreamStaker.unstakeToken()` subtracts `reward` from `totalRewardUnclaimed` without bounding it by the balance of the specific incentive being claimed. The relationship is indirectly enforced, since the value of `rewards` computed by `RewardMath.computeRewardAmount()` is lower than the total unclaimed balance, as long as `secondsInsideX128` is at or below `totalSecondsUnclaimedX128`. However, as the staker is compiled with Solidity 0.7.6, a violation could underflow `totalRewardUnclaimed` with potentially catastrophic effects.

Code Corrected

`SlipstreamStaker` and its libraries were removed in `Version 3`.

#028 SlipstreamStaker Constructor Misses Zero Address Checks

Informational Version 1

Code Corrected ✓

In `SlipstreamStaker`, the constructor rejects a zero `owner` but stores `factory`, `nonfungiblePositionManager` and `leafVoter` unchecked, and all three are immutable, so a zero value cannot be corrected after deployment.

Code Corrected

`SlipstreamStaker` and its libraries were removed in `Version 3`.

#029 withdrawToken Accepts the Zero Address as Recipient

Informational Version 1

Code Corrected ✓

In `SlipstreamStaker`, `withdrawToken()` rejects the staker itself as `to` but not the zero address, while `transferDeposit()` and `withdrawTokenFrom()` both reject it. A zero `to` reverts only because the position manager derives from OpenZeppelin's `ERC721`, whose `safeTransferFrom()` refuses a zero recipient.

Code Corrected

`SlipstreamStaker` and its libraries were removed in **Version 3**.

#031 CLGaugeTape Live Edge Reads Unvalidated Gauges

Informational Version 3

Acknowledged 

The NatSpec of `ICLGaugeTape.observe()` states that a contract that is not a CL gauge reverts. Nothing enforces this. `CLGaugeTape._live()` skips only addresses without code and queries any other address as a CL gauge:

```
if (_codeLength == 0) return _observation;
// ...
uint256 _effective = _clGauge.lastCumulativeRewardShare();
uint256 _projected = ILeafVoter(_clGauge.voter()).projectedCumulativeRewardShare(_gauge);
// ...
if (_projected > _effective) {
    if (_clGauge.pool().stakedLiquidity() > 0) _credited += _projected - _effective;
```

`observe()` takes the gauge address as an argument, so any contract can be passed to it. A contract exposing `lastCumulativeRewardShare()`, `voter()` and `pool()`, with `voter()` and `pool()` pointing at contracts of its choice, sets `cumulativeEffective` and `cumulativeCredited` of the live edge freely. Its far edge is zeroed, since only an allowlisted factory writes the rings. A consumer computing an APR over a gauge list it does not control thus gets attacker-chosen values instead of a revert.

Note that the `GaugeTape` of the MetaDEX repository avoids this by having `_live()` return a zeroed observation before touching the gauge when the gauge has no hourly observations stored. Because the rings are only written through an allowlisted factory, an address that was never recorded can never reach the external calls.

Acknowledged

Dromos Labs is aware of this issue but has decided to keep the code unchanged.

#032 NFTs Sent Directly to a Gauge Cannot Be Recovered

Informational Version 1

Acknowledged 

`NonfungiblePositionManager` treats a position as staked when `ownerOf(tokenId) == pool.gauge()`. A position NFT can reach the gauge without `deposit()`, through `transferFrom()`, through `safeTransferFrom()` (`CLGauge` inherits `ERC721Holder`), or through `mint()` with the gauge as recipient.

The gauge records no stake for such a token. `withdraw()` requires `_stakeOwner[tokenId] == account`, `deposit()` requires the caller to own the NFT, and the gauge never approves the token. The manager's `collect`, `decreaseLiquidity` and `burn` are restricted to the owner or an approved address, and `increaseLiquidity` requires the caller to be the gauge. `permit()` cannot be used, as `CLGauge` does not implement ERC-1271. Neither contract has a rescue path, so the NFT and the liquidity behind it are permanently locked.

Acknowledged

Dromos Labs is aware of this behavior and considers it to be in line with their design decisions:

We had some similar findings during the contest and we decided to acknowledge and treat it the same as transferring to any other type of uncontrolled address.

8 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.