SPEARBIT

Velodrome Finance Security Review

Auditors

D-Nice, Lead Security Researcher

Optimum, Lead Security Researcher

Alex The Entreprenerd, Security Researcher

Jeiwan, Associate Security Researcher

Report prepared by: Lucas Goiriz

January 22, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Velodrome Finance is a next-generation AMM that combines the best of Curve, Convex and Uniswap, designed to serve as Optimism's central liquidity hub. Velodrome NFTs vote on token emissions and receive incentives and fees generated by the protocol.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Velodrome according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 12 days in total, Velodrome Finance engaged with Spearbit to review the contracts, slipstream and universal-router protocols. In this period of time a total of **49** issues were found.

Summary

Project Name	Velodrome Finance
Repository	contracts, slipstream, universal-router
Commit	ba4b19...c64449, 9ab341...7c8c46, 5834d3...005601
Type of Project	DEX, DeFi
Audit Timeline	Nov 20 to Dec 5
Two week fix period	Dec 5 - Dec 19

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	6	6	0
Medium Risk	6	4	2
Low Risk	18	13	5
Gas Optimizations	9	8	1
Informational	10	8	2
Total	**49**	**39**	**10**

5 Findings

5.1 High Risk

5.1.1 `CLGauge` stakers can earn more than their due, with claimable rewards exceeding available rewards

Severity: High Risk

Context: CLPool.sol#L860-L865, CLGauge.sol#L120-L123, CLGauge.sol#L302-L306

Description: Staking users are intended to earn rewards submitted during an epoch. In case there was no staking, the rewards should be rolled over, and available to stakers in the next epoch where rewards are added.

This is currently handled in the code. However, in addition to the rollover, the current logic in `CLPool._updateRewardsGrowthGlobal` and `CLGauge._earned` can also retroactively apply rewards from prior epochs, with ideal scenarios resulting in a doubling of claimable rewards from prior periods.

This can occur from both malicious and benign actors. In case of a single party as a staker, they may be unable to withdraw their rewards, as their rewards would exceed the balance. A malicious actor may be able to withdraw the entire reward balance much earlier than they should, and then leave the Gauge in a perpetual overdrawn state thereafter. In case of multiple stakers, it would essentially be a race on emptying the rewards, with a number of stakers left with no rewards and no way to withdraw.

Recommendation: `CLGauge.left` is the implementation appropriately following the expected reward specfication, while the aforementioned two functions that should mimic its behavior are not consistent with it, nor consistent with each other currently, which they should be.

The recommended fix to resolve this issue has to do with the logical preconditions computing `timeDelta`:

```diff
+ uint256 _lastUpdatedEpoch = VelodromeTimeLibrary.epochStart(_lastUpdated);
- if (VelodromeTimeLibrary.epochStart(timestamp) != VelodromeTimeLibrary.epochStart(_lastUpdated)) {
+ if (periodFinish > _lastUpdatedEpoch && VelodromeTimeLibrary.epochStart(timestamp) >
↪   _lastUpdatedEpoch) {
      timeDelta = VelodromeTimeLibrary.epochNext(_lastUpdated) - _lastUpdated;
- } else if (periodFinish == VelodromeTimeLibrary.epochStart(timestamp)) {
+ } else if (periodFinish <= VelodromeTimeLibrary.epochStart(timestamp)) {
      // new epoch, notify has yet to be called so we skip the update
      timeDelta = 0;
  }
```

In their prior state, they would retroactively apply rewards due to non-zero `timeDelta` when the current epoch exceeded that of the last rewarded epoch or in any case where the current epoch didn't match the latest update.

These fixes ensure no retroactive rewards can be applied by making sure we apply a zero `timeDelta` in the case of being in an epoch with no applied rewards, and let the rollover mechanism in `CLGauge._notifyRewardAmount` re-applying prior missed rewards to the current epoch rewards.

These fixes should be applied to both `CLPool._updateRewardsGrowthGlobal` and `CLGauge._earned`.

Velodrome The recommended fix has been applied in commit 269eed7d (i.e. ensuring that `timeDelta` is not applied when periods do not match).

Spearbit: Recommended fixes appear applied in both recommended places and supporting test case added. *Posterior note*: this specific issue is now fixed by overhaul to the fee system, which is simpler than the prior logic that fixed this issue.

5.1.2 `SwapRouter` doesn't refund unspent ETH after swapping

Severity: High Risk

Context: SwapRouter.sol#L106, SwapRouter.sol#L126, SwapRouter.sol#L196, SwapRouter.sol#L220

Description: SwapRouter allows swapping of ETH for ERC20 tokens. The difference between selling ETH and an ERC20 token is that the contract can compute and request from the user the exact amount of ERC20 tokens to sell, but, when selling ETH, the user has to send the entire amount when making the call (i.e. before the actual amount was computed in the contract). As swaps made via `SwapRouter` can be partial, there are scenarios when ETH can be spent partially. However, the contract doesn't refund unspent ETH in such scenarios:

1. When `sqrtPriceLimitX96` is set (SwapRouter.sol#L80, SwapRouter.sol#L164), the swap will be interrupted when the limit price is reached, and some ETH can be left unspent.
2. A swap can be interrupted earlier when there's not enough liquidity in a pool.
3. Positive slippage can result in more efficient swaps, causing exact output swaps to leave some ETH unspent (even when it was pre-computed precisely by the caller).

As a result, `SwapRouter` can hold some leftover ETH after a swap was made. This ETH can be withdrawn by anyone via the `SwapRouter.refundETH()` function, causing a loss to the `SwapRouter` user.

Recommendation: In `SwapRouter.exactInputSingle()`, `SwapRouter.exactInput()`, `SwapRouter.exactOutputSingle()`, and `SwapRouter.exactOutput()` functions, consider returning unspent ETH to the caller at the end of the functions. The `PeripheryPayments.refundETH()` function can be used for that.

Velodrome: The recommended fix has been applied in commit 0c5da40e (i.e. a call to `refundETH()` will be made at the end of the swap functions inside `SwapRouter`).

Spearbit: Fixed as recommended in commit 0c5da40e.

5.1.3 Undistributed rewards are not rolled over when additional rewards are announced in an epoch

Severity: High Risk

Context: CLGauge.sol#L346-L350

Description: `CLGauge._notifyRewardAmount()` allows notifying rewards multiple times in an epoch. However, undistributed rewards are not rolled over during additional reward notifications:

1. When notifying rewards for the first time in an epoch, undistributed rewards are computed using `CLPool.timeNoStakedLiquidity()` and rolled over to the remaining time of the epoch (CLGauge.sol#L333-L343):

```
// rolling over stuck rewards from previous epoch (if any)
uint256 tnsl = pool.timeNoStakedLiquidity();
// we must only count tnsl for the previous epoch
if (tnsl + timeUntilNext > DURATION) {
    tnsl -= (DURATION - timeUntilNext); // subtract time in current epoch
    // skip epochs where no notify occurred, but account for case where no rewards
    // distributed over one full epoch (unlikely)
    if (tnsl != DURATION) tnsl %= DURATION;
}
_amount += tnsl * rewardRate;
rewardRate = _amount / timeUntilNext;
```

2. When notifying additional rewards in an epoch, undistributed rewards are not computed and not rolled over (CLGauge.sol#L346-L349):

```
uint256 _remaining = periodFinish - timestamp;
uint256 _leftover = _remaining * rewardRate;
IERC20(rewardToken).safeTransferFrom(_sender, address(this), _amount);
rewardRate = (_amount + _leftover) / timeUntilNext;
```

The two branches then call `CLPool.syncReward()` (CLGauge.sol#L344, CLGauge.sol#L350), which resets the value of `timeNoStakedLiquidity` (CLPool.sol#L891). Since the second branch doesn't roll over the undistributed rewards (which are computed using the value of `CLPool.timeNoStakedLiquidity`), they'll remain locked in the contract. Due to the way rewards are notified (new rewards are always added to the remaining ones), these rewards cannot be unlocked by new notifications.

Recommendation: Consider always rolling over the undistributed rewards (i.e. the rewards issued during the duration of `timeNoStakedLiquidity`) when notifying new rewards.

Velodrome: The fix has been implemented in commit 3ee4b109 and will rollover rewards regardless of whether or not notify reward amount is called with or without claim.

Spearbit: Fixed as recommended in commit 3ee4b109.

5.1.4 Reward notification can unnecessarily reduce `tnsl`, causing locking of rewards

Severity: High Risk

Context: CLGauge.sol#L336

Description: The CLPool.timeNoStakedLiquidity variable tracks the duration when there was a reward in a pool but no staked liquidity. The variable can be updated in epochs following the one it's supposed to track, thus `CLGauge._notifyRewardAmount()` needs to reduce its value to find the actual period of no staked liquidity in the previous epoch:

```
uint256 timeUntilNext = VelodromeTimeLibrary.epochNext(timestamp) - timestamp;
// ...
// we must only count tnsl for the previous epoch
if (tnsl + timeUntilNext > DURATION) {
    tnsl -= (DURATION - timeUntilNext); // subtract time in current epoch
    // ...
```

However, it incorrectly detects the length of `timeNoStakedLiquidity` outside of its epoch: in the snipped above, the remaining time of the current epoch is added to `tnsl`; instead, the passed time in the current epoch should be added.

As a result, `tnsl` will be mistakenly reduced when it shouldn't. Since `tnsl` is used to compute the amount of rewards to roll over from the previous epoch (CLGauge.sol#L342), a mistakenly reduced `tnsl` will reduce the amount of rewards that will be rolled over. Because of the way rewards notification works (new rewards are always added to the current ones), the amount of rewards that wasn't rolled over due to the unnecessary reduction will remain locked in the contract.

Recommendation: To fix the issue directly, consider this change:

```
--- a/contracts/gauge/CLGauge.sol
+++ b/contracts/gauge/CLGauge.sol
@@ -333,7 +333,7 @@ contract CLGauge is ICLGauge, ERC721Holder, ReentrancyGuard {
             // rolling over stuck rewards from previous epoch (if any)
             uint256 tnsl = pool.timeNoStakedLiquidity();
             // we must only count tnsl for the previous epoch
-            if (tnsl + timeUntilNext > DURATION) {
+            if (tnsl + (DURATION - timeUntilNext) > DURATION) {
                 tnsl -= (DURATION - timeUntilNext); // subtract time in current epoch
                 // skip epochs where no notify occurred, but account for case where no rewards
                 // distributed over one full epoch (unlikely)
```

However, to reduce the complexity and avoid other issues caused by `tnsl` miscalculations, we recommend the following change in the `CLPool` contract:

```
--- a/contracts/core/CLPool.sol
+++ b/contracts/core/CLPool.sol
@@ -869,6 +869,9 @@ contract CLPool is ICLPool {
                     rewardReserve -= reward;
                     rewardGrowthGlobalX128 += FullMath.mulDiv(reward, FixedPoint128.Q128,
↪   stakedLiquidity);
                 } else {
+                    if (timestamp >= periodFinish && _lastUpdated < periodFinish) {
+                        timeDelta = periodFinish - _lastUpdated;
+                    }
                     // gauge can compute a different value and use that instead of the actual,
                     // depending on unstaked time crossing epoch flips
                     timeNoStakedLiquidity += uint32(timeDelta);
```

I.e. we recommend limiting the growth of `timeNoStakedLiquidity` so that it never goes beyond the current epoch. This will allow to remove the `tnsl` calculation logic from `CLGauge._notifyRewardAmount()`.

Velodrome: Fixed in commit a829b218.

Spearbit: Verified. Previously, the math was accounting for time in which no staked liquidity was deposited

The math was logically the following:

- If in same epoch → relative delta.
- If in new epoch → Shifted relative delta.

The shifted relative delta had 2 implications:

- In scenarios in which more than one epoch was passed, the time calculated was above the entire `DURATION`.
- In scenarios in which less than an epoch was passed, then some rewards may have been stuck due the issue with synchronization.

The new code works as follows:

- Time and rewards always pass, no matter what.
- If not enough time has passed → no issue happens as the `notifyRewardAmount` logic already accounts for this.
- if too much time has passed → the amount is limited by `if (reward > rewardReserve) reward = rewardReserve;`.

This makes it so that the `rewardRate` is always dripped, no matter the amount of time in which no staked liquidity was deposited.

Those rewards are re-queued, the only additional problem with this approach can be a marginal loss of dust for amounts below `DURATION`, this is not a new issue but a known problem with SNX-like reward systems that compress the total rewards due to using `rewardsPerSeconds`.

The fix has been implemented by drastically simplifying the Pool to:

- Always spend the rewards.
- Stop once all rewards are dripped no matter what.

And the Gauge to:

- Check the `rollover` (the rewards dripped to no staked liquidity).
- Requeue the `rollover`.

5.1.5 `CLGauge.deposit()` doesn't verify that the liquidity to stake is from the pool the gauge integrates with

Severity: High Risk

Context: CLGauge.sol#L160

Description: `CLGauge.deposit()` allows to stake an NFT that provides liquidity to a different pool (i.e. a pool the gauge doesn't integrate with). This allows a malicious actor to stake liquidity into a `CLGauge` from a fake `CLPool` and earn gauge rewards, stealing them from the liquidity providers of the gauge's pool.

Recommendation: In `CLGauge.deposit()`, consider ensuring that the `tokenId` provides liquidity in the `pool`:

1. First, `CLGauge` needs to know the address of the `CLFactory` that deployed the `pool`.
2. Then, the values of `token0`, `token1`, and `tickSpacing` of the staked NFT's pool can be read from `NonfungiblePositionManager.positions()`.
3. Finally, the pool address can be computed using `PoolAddress.computeAddress()` and compared to the value of `pool`.

Velodrome: The issue has been fixed in commit b3b0f8a1.

As the `NonfungiblePositionManager`, `CLPoolFactory` and `CLGauge`'s are bound to one another in a way that is immutable post deployment, the following alternative fix was implemented:

- Adding a new storage variable to `CLGauge` that caches the pool's `tickSpacing`.
- On deposit, check that the position's `token0`, `token1` and `tickSpacing` values match with the same values stored in the gauge, with a revert otherwise.

Spearbit: Fixed in commit b3b0f8a1.

5.1.6 Liquidity staked at ticks can be manipulated to impact swap fees and rewards

Severity: High Risk

Context: CLPool.sol#L544-L545, CLPool.sol#L367-L375

Description: When removing all staked liquidity from a position via `CLGauge.decreaseStakedLiquidity()`, the amounts of staked liquidity at the ticks of the position are recorded incorrectly: due to the clearing of the ticks, the lower tick will have the negative amount of staked liquidity and the upper tick will have the positive amount. In a normal situation, both values should be 0.

`CLGauge.decreaseStakedLiquidity()` calls `NonfungiblePositionManager.decreaseLiquidity()` (CLGauge.sol#L270) to remove liquidity from the position. The latter calls `CLPool.burn()` (NonfungiblePositionManager.sol#L296) to burn the liquidity from the pool. Burning liquidity updates a position (CLPool.sol#L497-L504, CLPool.sol#L260), which clears the ticks if all liquidity was removed from the position (CLPool.sol#L367-L375). This clearing will also clear the values of stakedLiquidityNet of the ticks. However, `CLGauge.decreaseStakedLiquidity()` then calls `CLPool.stake()` (CLGauge.sol#L280), which updates the staked liquidity in the ticks that have just been cleared (CLPool.sol#L544-L545).

As a result, the total amount of staked liquidity in a pool can be skewed during swapping as pool's staked liquidity is updated with the values stored at ticks (CLPool.sol#L726). This will have a strong impact on the protocol since both swap fees (CLPool.sol#L684-L685) and gauge rewards (CLPool.sol#L870) are computed based on the amount of staked liquidity.

While the impact can be caused by any user who removes their liquidity via `CLGauge.decreaseStakedLiquidity()`, we also believe that the miscalculation can be triggered intentionally to manipulate the value of staked liquidity to gain benefits from swap fees and/or gauge rewards.

Recommendation: In `CLPool.stake()`, consider skipping a call to `ticks.updateStake()` when the value of `initialized` of the respective tick is `false`.

Velodrome: The issue has been fixed in commit 171a82f2.

The fix that has been applied is to skip calls to `updateStake()` when the tick has been cleared (i.e. is uninitialized). This should only occur on a call to `decreaseStakedLiquidity()` when the following are true:

- The entire liquidity in the position is decreased.
- The liquidity is the only liquidity in the given tick ranges.

Spearbit: Fixed as recommended in commit 171a82f2.

5.2 Medium Risk

5.2.1 `UniswapV2Library.getAmountIn` uses hardcoded fees, but Velodrome Pools have custom fees

Severity: Medium Risk

Context: UniswapV2Library.sol#L121-L132

Description: `getAmountIn` uses fees to determine the `amountIn` necessary to receive a given `amountOut` (see UniswapV2Library.sol#L121-L132):

```
function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, Route memory route)
    internal
    view
    returns (uint256 amountIn)
{
    if (reserveIn == 0 || reserveOut == 0) revert InvalidReserves();
    if (!route.stable) {
        amountIn = (amountOut * 1000 * reserveIn) / ((reserveOut - amountOut) * 997) + 1;
    } else {
        revert StableExactOutputUnsupported();
    }
}
```

This is cognizant of fees (997), but the fees are hardcoded, which may cause the math to be incorrect, since `Pool` allows custom fees to be set via the factory (see Pool.sol#L372-L373):

```
if (amount0In > 0) _update0((amount0In * IPoolFactory(factory).getFee(address(this), stable)) / 10000);
↪  // accrue fees for token0 and move them out of pool
if (amount1In > 0) _update1((amount1In * IPoolFactory(factory).getFee(address(this), stable)) / 10000);
```

Recommendation: Consider whether the router needs to support all pools, and add a functionality to adapt to custom fees.

Velodrome: Support for custom fees on v2 pools has been added in commit f6185838.

Spearbit: Fixed by passing `pair` and `factory` to `getAmountIn` and `getAmountOut`.

5.2.2 Implementations of clones could be metamorphic and lead to exploit

Severity: Medium Risk

Context: CLGaugeFactory.sol#L20, CLFactory.sol#L49, PoolFactory.sol#L35

Description: `CLPool`, `CLGauge` and `Pool` contracts are deployed utilizing OZ `Clones` library. It is defined as

> A library that can deploy cheap minimal non-upgradeable proxies

The proxies themselves will always be locked to a specific implementation, and in that sense cannot be upgraded, but the code at the implementation address could change, by what's often referred to as "*metamorphic smart contracts*".

This opens a variety of possible options. A full blown metamorphic contract could lead to complete compromise of the contracts via many scenarios. The one saving grace, is that it requires a malicious party in control of the deploy pipeline to execute this (dev). But in the case of such a scenario, one simple but compromising attack would be a metamorphic CLGauge implementation that initially calls nft.setApproveForAll on behalf of a malicious operator, giving approval for any future Pool NFTs it would own.

This could be done prior to any actual staking to avoid detection. Once successfully executed across the gauges expected to have highest staking, the implementation could be metamorphised to the non-exploitative contract, missing that functionality (the fact that this contract cannot be exploited doesn't matter, as the exploit has already run and is present on the NFT contract, regardless of implementation's contents). Staking users could audit the current implementation themselves and would not see this possibility in the current code, and assume it is safe. However, any liquidity they stake, could be withdrawn, as the malicious operator would have approval to send the staked NFTs to any address they wish, and subsequently remove their liquidity and tokens from the pool to their own controlled addresses.

A simpler risk that could make its way in simply by error would be an implementation that contains the SELFDESTRUCT opcode or a DELEGATECALL which could lead access to the code. In this case any party could brick all the proxies dependent on the implementation in question, resulting in permanent fund loss, unless CREATE2 was used to deploy the implementation (in which case we return full circle to a metamorphic contract).

Recommendation: Consider hardcoding implementation to an address which when checked with a tool like metamorphic-contract-detector returns false for conditions 1, 2 and 6. There are ways to obfuscate metamorphic contracts, and if 3, 4 and 5 are positive it may warrant further investigation even if the former are false.

If a hardcoded option is not considered ideal, an option is to include the codehash that can be resolved to an address which ideally passes the above checks, and including logic that the implementation being set in the constructor has that same codehash, so we're not bound to just a specific address.

End-users are encouraged to do these checks on the implementations themselves prior to interaction, albeit following the proper implementation of the recommendations should minimize the risk and ideally devops of the team would have an internal multi-party pipeline to check for this.

Velodrome: The issue has been acknowledged and we will consider option 1 or 2 depending on whether or not we can easily modify the deployment pipeline without a degradation in the development experience.

Spearbit: Acknowledged.

5.2.3 GovernorSimple and VetoGovernor minimum weight math uses the incorrect divisor

Severity: Medium Risk

Context: VetoGovernor.sol#L633-L634, GovernorSimple#L588-L589

Description: The variable COMMENT_DENOMINATOR is defined in both GovernorSimple and VetoGovernor to express a ratio of 4BPS:

- VetoGovernor.sol#L48-L49:

```
uint256 public constant override COMMENT_DENOMINATOR = 1_000_000_000;
```

- VetoGovernor.sol#L633-L634:

```
uint256 minimumWeight = (escrow.getPastTotalSupply(startTime) * commentWeighting) / 10_000; /// @audit
↪  Incorrect denominator, should use `COMMENT_DENOMINATOR`
```

However, the denominator is left hardcoded as 10_000, which would change the minimum comment weight to 40% of the total supply.

Recommendation: Consider changing the math on commentWeighting to be in BPS, or use the COMMENT_DENOMINATOR as it seems that it was intended.

Velodrome: Fixed in commit 88861783 by using COMMENT_DENOMINATOR.

Spearbit: Verified.

5.2.4 Forwarder Gas Grief Still Possible if the `req.gas` is too close to the actual amount needed

Severity: Medium Risk

Context: Deployed Velodrome Forwarder

Description: The Velodrome Forwarder code checks for `gasLeft()` and then performs a set of operations, which has a cost that is not computed. That gas cost will directly impact the post EIP-150 `gasleft` that will be available for the recipient call.

Recommendation: Currently, requiring 5/10% more gas than necessary should be sufficient to prevent this as the range of values for the grief is very small.

See the latest version, which I believe is safe.

It is worth investigating if any griefing has happened, and solve it at the "*social layer*", by ensuring people require 10% above the gas necessary.

Long term, you should be safe to use OZs newer version of the relayer which I beleive is not subject to the grief since it checks the gas used (after the fact and after overheads) instead of the gas left (before the overheads, way more likely to be wrong)

Another solution, that Optimism has adopted is to have an irrationally big overhead for math, even when no value is sent (40k) this avoids the math errors in every case.

Velodrome: The issue has been acknowledged and we will partially implement the recommendation. We will check for instances of griefing and will assess the necessity of metatx feature. Longer term, we will either remove the dependency or implement the OZ relayer.

Spearbit: Acknowledged.

5.2.5 Incorrect result rounding in `UniswapV2Library.getAmountIn()`

Severity: Medium Risk

Context: UniswapV2Library.sol#L128, UniswapV2Library.sol#L100

Description: `UniswapV2Library.getAmountOut()` and `UniswapV2Library.getAmountIn()` are used to compute output and input amounts for swapping, respectively. By design (i.e. in Uniswap 2), these functions are counterparties to each other: an input amount should have only one respective output amount (given that pool reserves and the swap fee don't change), and vice versa. However, the `UniswapV2Library` contract computes swap fee amount differently in the functions:

1. In `UniswapV2Library.getAmountOut()`, the fee is applied to the input amount before the output amount is calculated (UniswapV2Library.sol#L100). Notice that the fee amount (`amountIn * 3 / 1000`) is subtracted from the input amount–this results in an amount that's rounded up (the division rounds down, and thus the difference will be rounded up). In Uniswap V2, however, the result is always rounded down (UniswapV2Library.sol#L49).
2. `UniswapV2Library.getAmountIn()` is identical to Uniswap V2 (UniswapV2Library.sol#L56-L58). Thus, the result is rounded down.

The difference in rounding in the two functions will impact "*exact output*" swaps: in some scenarios, the actual output amount will be greater than the one requested by the user by 1 wei; the input amount will be greater by 1 wei as well. Due to the maximum input amount check (V2SwapRouter.sol#L98), some "*exact output*" swaps can fail.

Recommendation: Consider the following change:

```
--- a/contracts/modules/uniswap/v2/UniswapV2Library.sol
+++ b/contracts/modules/uniswap/v2/UniswapV2Library.sol
@@ -125,7 +125,8 @@ library UniswapV2Library {
     {
         if (reserveIn == 0 || reserveOut == 0) revert InvalidReserves();
         if (!route.stable) {
-            amountIn = (amountOut * 1000 * reserveIn) / ((reserveOut - amountOut) * 997) + 1;
+            amountIn = (amountOut * reserveIn) / (reserveOut - amountOut);
+            amountIn = (amountIn * 1000) / 997 + 1;
         } else {
             revert StableExactOutputUnsupported();
         }
```

Velodrome: The recommended fix to make the fee application consistent across both the universal router and the pools has been applied in commit 44ca157f.

Spearbit: Fixed as recommended in commit 44ca157f.

5.2.6 NFT cannot be withdrawn after all liquidity was removed from the position

Severity: Medium Risk

Context: CLGauge.sol#L280, CLPool.sol#L539, Position.sol#L55

Description: `CLGauge.withdraw()` fails when all liquidity from a position has been removed using `CLGauge.decreaseStakedLiquidity()`. The NFT position remains locked in the `Gauge` contract and cannot be removed until some new liquidity is provided.

The error happens because `Position.update()` fails when a position has 0 liquidity and the `liquidityDelta` is also 0:

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

Calling `CLGauge.withdraw()` on an empty position tiggers `CLPool.stake()` with the value of `stakedLiquidity-Delta` equal 0, which calls `update()` on the empty Gauge's position and also passes 0 in the `liquidityDelta` parameter.

Recommendation: In `CLGauge.withdraw()`, consider skipping the call to `pool.stake()` when `liquidityToStake` is 0.

Velodrome: The recommended fix has been applied in commit 5c0eae52 (i.e. if the position is empty, a call to `pool.stake()` will be skipped).

Spearbit: Fixed as recommended in commit 5c0eae52.

5.3 Low Risk

5.3.1 `CLPool` reward system presents risks in the case of deploying on a chain with a Mempool

Severity: Low Risk

Context: CLPool.sol#L708-L709

Description: The rewards math on `CLPool` is based on the idea that rewards are dripped each second exclusively to the active liquidity. A drip of rewards increases the `rewardsGrowthGlobal` crediting said growth to all active liquidity.

The way this is done, in a swap, is by calling `_updateRewardsGrowthGlobal` on crossing ticks.

See CLPool.sol#L708-L709:

```
_updateRewardsGrowthGlobal(); /// @audit Called every time corssing happens
```

It's worth noting that this accrual process is based on a delta time, meaning that it ignores changes that happen within the same block (see CLPool.sol#L853-L857):

```
uint32 timestamp = _blockTimestamp();
uint256 _lastUpdated = lastUpdated;
uint256 timeDelta = timestamp - _lastUpdated;

if (timeDelta != 0) {
    // ...
```

In a mempool system, nothing happens between blocks: all of the action happens inside of the block. Since accrual can be triggered at the start of the block, the most optimal LP strategy would be to claim said rewards, then unstake the liquidity and JIT said liquidity to end users as a means to gain swap fees.

Liquidity Provision Mempool risks

In the case of a deployment on a chain with a mempool, bundling would allow Active LP to:

- Claim rewards until now.
- Unstake.
- JIT LP to gain swap fees.
- Restake at the end of block, to farm the new drip of rewards.

Ultimately this is a challenge in balancing effective volume against time spent in the pool. With the main issue being that, since actions / volume within a block is not counted, said gaming could be performed

This will be negative for Passive LPs but neutral for end users.

An interesting edge case of the above, meant to allow this on non-mempool system could be “wrapped swaps”, which would ensure a swap is MEVd back and forth as a way for the LP to ensure that their liquidity remains in the active range, this may be positive for end-users, and indifferent to passive LPs.

Recommendation: Monitor the behaviour of Active vs Passive LPs and users and determine if a Volume Based reward system would be fairer vs a time based system which may be gameable, but gaming it may be too expensive.

Velodrome: The issue has been acknowledged and no further action will take place. Monitoring of LP behavior will take place if a mempool exists.

Spearbit: Acknowledged.

5.3.2 `unstakedFee` may be paid for Pools with a killed gauge or that will never receive any votes

Severity: Low Risk

Context: CLPool.sol#L923-L924

Description: All pool fees apply the `splitFees` formula which uses `applyUnstakedFees`, which will query for the `unstakedFee` (see CLPool.sol#L923-L924):

```
uint256 _stakedFee = FullMath.mulDivRoundingUp(_unstakedFeeAmount, unstakedFee(), 1_000_000);
```

Which will default to (see CLFactory.sol#L157-L164):

```
function getUnstakedFee(address pool) external view override returns (uint24) {
    if (unstakedFeeModule != address(0)) {
        return IFeeModule(unstakedFeeModule).getFee(pool);
    } else {
        // Default unstaked fee is 10%
        return 100_000;
    }
}
```

This ultimately means that it's highly likely for all pools to always charge a 10% unstake fee. In the case of a pool that shouldn't receive votes, the loss would be marginal and should be "self correcting":

- LPs will receive 10% lower fees.
- Eventually some LPs will decide to vote to redistribute those fees as voting rewards.
- This will trigger a "race" to vote and lock in that extra 10%.

This dynamic should reach equilibrium and should also trigger people to engage with the system, at the cost of a less straightforward LPs proceess to gain fees. In the case of a killed Gauge however, the Voter will prevent calling `notifyRewardAmount`, and nobody will be able to vote on the Pool (see Voter.sol#L487-L497):

```
function _distribute(address _gauge) internal {
    _updateFor(_gauge); // should set claimable to 0 if killed
    uint256 _claimable = claimable[_gauge];
    if (_claimable > IGauge(_gauge).left() && _claimable > DURATION) {
        claimable[_gauge] = 0;
        IERC20(rewardToken).safeApprove(_gauge, _claimable);
        IGauge(_gauge).notifyRewardAmount(_claimable);
        IERC20(rewardToken).safeApprove(_gauge, 0);
        emit DistributeReward(_msgSender(), _gauge, _claimable);
    }
}
```

In those cases, the `unstakeFee` will be lost, and recovering it would create a scenario where the pool would be voted on again (which may be problematic if the Gauge was killed to prevent economic or governance attacks). Meaning that in those cases, the Pool will be underpaying swap fees by 10% and those fees will not be recoverable.

This can be avoided by ensuring that the `unstakeFee` is zero when the Gauge is killed, which should be doable by the `swapFeeManager` and should be done in synch with killing the gauge.

Recommendation: Ensure that a Pool with a killed Gauge has a 0 `unstakeFee`.

Velodrome: The recommended fix has been applied in commit eb832564, where it slightly modifies the `getUnstakedFee` function to `return 0` if the gauge is not alive. Changed the function to:

```
function getUnstakedFee(address pool) external view override returns (uint24) {
    if (!IVoter(voter).isAlive(ICLPool(_pool).gauge())) {
        return 0;
    }
    if (unstakedFeeModule != address(0)) {
        return IFeeModule(unstakedFeeModule).getFee(pool);
    } else {
       // Default unstaked fee is 10%
       return 100_000;
    }
}
```

Spearbit: The issue has been fixed by checking if the Gauge is alive in the voter. If the gauge is not alive, then the `unstakeFee` will default to 0. Verified.

5.3.3 Inconsistent size for timestamp results in eventual desync

Severity: Low Risk

Context: CLPool.sol#L862, CLGauge.sol#L120

Description: The `CLPool` contract works with timestamps as a `uint32` type which wouldn't overflow until the year 2106 and allows for gas savings with respect to storage costs. The `CLGauge` and `VelodromeTimeLibrary` uses `uint256` for timestamps, which has the advantage of generally lesser runtime gas overhead and no practical worry of overflow. `CLPool` interacts with this contract and library, and has reliance on some of their timestamps and calculations.

At the overflow point, the timestamps will desync from each other, and some prior functionality will no longer work as expected, with CLPool.sol#L862 being an example of this.

Recommendation: Use a consistent size for timestamps and durations across a design architecture if possible, and especially within the same contract to avoid eventual desync issues. The `CLPool` design appears to be valid for up to the year 2106, and that should be accounted for. Additionally by switching the type down to 32 bits for `periodFinish` further gas storage gains may be realized and desync issues minimized.

Velodrome: Acknowledged but will leave unfixed, but have noted that the design is only valid until the year 2106.

Spearbit: Acknowledged.

5.3.4 Outdated solidity compiler versions are being used

Severity: Low Risk

Context: General scope

Description: The repository is using compiler versions that are less than `0.8.0` which is not recommended. Newer versions incorporate enhanced security features, addressing vulnerabilities present in older releases. In the specific case of pragma versions `>= 0.8.0` arithmetic operations are inherently safe which will naturally decrease the attack surface for this repository.

Recommendation: Consider bumping the pragma version to `>= 0.8.0`. For more information please refer to Solidity v0.8.0 Breaking Changes.

Velodrome: Acknowledged. We opted to stick with v0.7.6 across both pools and gauges to take advantage of the security guarantees provided by UniswapV3's pools.

Spearbit: Acknowledged.

5.3.5 Missing validation check for `unstakedFee` in `CLPool.applyUnstakedFees/unstakedFee`, which might be more than 100%

Severity: Low Risk

Context: CLPool.sol#L919

Description: `unstakedFee` returns a value that is used in the context of percentage, i.e. there is an implicit assumption that the return value should be <= `1_000_000` while in reality this is not enforced in the code and thus may cause an underflow in the value of `unstakedFeeAmount`, which will totally corrupt the values of `feeGrowthGlobal0X128`, `feeGrowthGlobal1X128`, `gaugeFees.token0`, `gaugeFees.token1`.

Recommendation: Consider adding a validation check to ensure `unstakedFee` returns a value that is <= `1_000_000`, otherwise either revert or use a default value instead.

Velodrome: The fix has been applied in commit 4baded78 and now `getUnstakedFee()` cannot return values greater than `1_000_000`.

Spearbit: Fixed in commit 4baded78 by implementing the recommendation and adding a default value.

5.3.6 `getSwapFee` and `getUnstakedFee` may return a value above their intended limits

Severity: Low Risk

Context: CLFactory.sol#L167-L169

Description: `enableTickSpacing` caps the max fee at `100_000` pips (10%). `getUnstakedFee` by default returns `100_000` and implicitly should never return any value above `1_000_000` pips.

See CLFactory.sol#L167-L169:

```
function enableTickSpacing(int24 tickSpacing, uint24 fee) public override {
    require(msg.sender == owner);
    require(fee <= 100_000);
    // ...
```

In a scenario of malicious governance, either wanting to brick a pool or wanting to grief end users, `swapFeeModule` and `unstakedFeeModule` could be set in a way that would bypass the intended limits for `getSwapFee` and `getUnstakedFee`.

Proof of Concept

- Call `setSwapFeeModule`and `setSwapFeeManager` with an implementation that returns any value.
- The pool will use said value, resulting in bricked functionality, or substantial loss to end users.

Recommendation: Ensure that the return value from both functions is capped to the maximum value acceptable as to strengthen onchain guarantees.

Velodrome: The recommended fix has been applied in commit 4baded78 and now the return values from both `getSwapFee()` and `getUnstakedFee()` functions cannot exceed their maximum allowed values.

Spearbit: Verified. Fixed by limiting or default to a safe value.

5.3.7 `NonfungiblePositionManager` may cause losses if the `tokensOwed0` overflow

Severity: Low Risk

Context: NonfungiblePositionManager.sol#L308-L322

Description: The casting of `amount0` from `uint256` to `uint128` as well as the sum of `tokensOwed` overflow can cause a loss to users of the `NonfungiblePositionManager` for tokens that have very high decimals (see NonfungiblePositionManager.sol#L308-L322):

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

if (!isStaked) {
    position.tokensOwed0 += uint128(
        FullMath.mulDiv(
            feeGrowthInside0LastX128 - position.feeGrowthInside0LastX128, positionLiquidity,
↪   FixedPoint128.Q128
        )
    );
    position.tokensOwed1 += uint128(
        FullMath.mulDiv(
            feeGrowthInside1LastX128 - position.feeGrowthInside1LastX128, positionLiquidity,
↪   FixedPoint128.Q128
        )
    );
}
```

The amount of tokens necessary for the loss is `3.4028237e+38`. This is equivalent to `1e20` value with 18 decimals. This is not a new bug but a risk with the UniV3 implementation.

Recommendation: The bug doesn't require a fix but it's recommended to document this risk to end users.

Velodrome: Acknowledged, and documentation has been updated.

Spearbit: Acknowledged.

5.3.8 `Pool` First Depositor Imbalanced LPing for Stable Pool may cause loss to 2nd depositor

Severity: Low Risk

Context: Pool.sol#L300-L301

Description: Stable Pools are seeded with the assumption that both tokens will be deposited at a 1:1 ratio.

The Router `quoteLiquidity` function has the following comment (see Router.sol#L81-L82):

```
/// @dev this only accounts for volatile pools and may return insufficient liquidity for stable pools
function quoteLiquidity(
    // ...
```

Meaning that it can be incorrect for Stable Pools. The `Sync` function allows to update reserves without minting tokens (see Pool.sol#L392-L394):

```
function sync() external nonReentrant {
    _update(IERC20(token0).balanceOf(address(this)), IERC20(token1).balanceOf(address(this)), reserve0,
↪   reserve1);
}
```

The first deposit on a Stable Pool has got to be 1:1 and will mint the LP token at a 1:1 ratio (see Pool.sol#L307-L314):

```
uint256 _totalSupply = totalSupply(); // gas savings, must be defined here since totalSupply can update
↪   in _mintFee
if (_totalSupply == 0) {
    liquidity = Math.sqrt(_amount0 * _amount1) - MINIMUM_LIQUIDITY;
    _mint(address(1), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens -
↪   cannot be address(0)
    if (stable) {
        if ((_amount0 * 1e18) / decimals0 != (_amount1 * 1e18) / decimals1) revert DepositsNotEqual();
        if (_k(_amount0, _amount1) <= MINIMUM_K) revert BelowMinimumK();
    }
    // ...
```

By using `sync` the first depositor is able to break this expectation. The first depositor can:

- Donate imbalanced (Transfer tokens directly).
- Sync (update reserves).
- Mint with a small amount of tokens.
- This will have rebased the LP token value (protected against donation by the min liquidity size).
- This will have imbalanced the pool, while giving 100% of the underlying value to the first minter.
- Subsequent depositors, that deposit in a balanced way, will lose some of their value due to the min math which requires a pro-rata contribution.

(See Pool.sol#L316-L317):

```
liquidity = Math.min((_amount0 * _totalSupply) / _reserve0, (_amount1 * _totalSupply) / _reserve1);
```

A more likely scenario will be that smart projects will be able to seed pools at a depegged rate, and in some edge case some people will lose a % of the fair value of their LP tokens due to incorrect slippage checks (as they will assume the pool will accept at a 1:1 or close to 1:1 rate).

Recommendation: It may be best to prevent `sync`ing when there's 0 liquidity in the pool. Additionally, this attack carries risk for the initial depositor, as they are imbalancing the pool, one of the two tokens will be discounted. Ensuring a `minLiquidity` check when lping should prevent this scenario.

Velodrome: The recommended fix has been applied in commit 34af3b94 by adding a `minLiquidity` check while lping and ensuring that sync cannot be called while the Pool has no liquidity.

Spearbit: Fixed by preventing `sync` on 0 liquidity. Verified.

5.3.9 `CLPool.collectFees` can result in unnecessary 0 value transfers

Severity: Low Risk

Context: CLPool.sol#L948-L964

Description: If one of the `gaugeFees` tokens amounts to 1, it would result in a 0 value token transfer being fired off, due to the logic that tries to ensure the fee tokens never have their storage zeroed. In the normal case, this would yield unnecessary additional gas costs and events indicating 0-value transfers occurred. In the case of a non-standard ERC-20 token, it could result in the call to this function failing and reverting, thereby blocking the function and possibly the other non-zero token from being claimed until this condition is resolved.

Recommendation: Due to the logic in question, which decrements the actual collected fees by 1, the amount checks should not be `> 0` but `> 1`:

```
- if (amount0 > 0) {
+ if (amount0 > 1) {
      // ...
```

similar should be done for the `amount1` branch.

Velodrome: The recommended fix has been applied in commit 41d0e9af (`amounts` should now be greater than 1 and not 0).

Spearbit: Verified that commit 41d0e9af implements the recommendation.

5.3.10 Factory setters intended to be callable once are not limited to a single successful call

Severity: Low Risk

Context: CLFactory.sol#L191-L204, ICLFactory.sol#L174-L183, CLFactory.sol#L43-L45, CLGaugeFactory.sol#L14-L28, ICLGaugeFactory.sol#L14-L17

Description: The above setters, such as `CLFactory.setNonfungiblePositionManager` are documented to be settable only once. This is not the case, and each setter could be called multiple times with the right conditions. The current code design assumes it will be passed off to a contract that doesn't support calling that function again, however, it does inherit the permissions to indeed call and set again.

Recommendation: These setters should only be callable once and rather than relying on assumptions and discounting non-ideal cases, the setters could be made single call without assumptions in all cases. The following changes in `CLFactory.setNonfungiblePositionManager` would accomplish this:

```diff
 function setNonfungiblePositionManager(address _nft) external override {
-     require(nft == msg.sender, "AI");
+     require(nft == address(0), "AI");
+     require(owner == msg.sender, "NA");
      require(_nft != address(0));
      nft = _nft;
 }
```

instead of unnecessarily using `nft` as a placeholder for `owner`, we just check the owner, and make sure `nft` is in fact unset, before setting it. This would make the initial setting of `nft` to the `owner` in the constructor unnecessary. The recommended fix as written would also pass the current test cases (necessitating a minor update on expected revert message for one case from `AI` to `NA` which is more correct anyways).

Additionally, this could allow further safeguards on `createPool` and similar side effect functions that depend on a proper `nft` being set. In the earlier case, checking this is complicated, while with this change, an additonal non-zero check on `nft` can be introduced prior to `createPool` completing.

Similar changes are recommended for `CLFactory.setGaugeFactory` and `CLGaugeFactory.setNonfungiblePositionManager`. The latter would require additional inclusion of an `owner` state variable rather than piggybacking the `nft` variable for that purpose.

Velodrome: The recommended fix has been applied in commit 4baded78 by ensuring that the addresses to be updated by setters have never been initialized and that `msg.sender` is the deployer.

Spearbit: Verified that commit 4baded78 implements the recommendation.

5.3.11 `IFeeModule` could be called via `ExcessivelySafeCall` and limited gas to avoid bricking pools in case of governance attack

Severity: Low Risk

Context: CLFactory.sol#L148-L164

Description: While the risk is very low, the current implementation of `getSwapFee` and `getUnstakedFee` calls a target, forwards all of it's available gas and will bubble a revert up to the caller. This would allow, by jumping multiple governance hoops, to brick a pool via the `FeeModule`.

While a `try/catch` may seem like a valid mitigation, this would still leave the contract vulnerable to return bombs. It's worth noting, that in such a scenario people would still be able to withdraw their LP tokens as `fees` are computed only in `swap` and `flash`

Recommendation: The safest way to allow the integration of an untrusted module is to use ExcessivelySafeCall with capped gas (e.g. 200k gas, extremely safe) which would enforce no higher consumption as well as avoid malicious reverts and return bombs.

Velodrome: The recommended fix has been applied in commit 4fedacfb by using the `ExcessivelySafeCall` module.

Spearbit: Verified.

5.3.12 Increased surface area for potential future reentrancy attacks in `SwapRouter` and `NonfungiblePositionManager`

Severity: Low Risk

Context: PeripheryPayments.sol#L30, PeripheryPayments.sol#L40, PeripheryPayments.sol#L19

Description: The `PeripheryPayments` contract has 3 external functions allowing anyone to claim any excess tokens/eth from the contract: `unwrapWETH9`, `sweepToken`, `refundETH`. The implicit assumption is that contracts that inherit this functionality like `SwapRouter` and `NonfungiblePositionManager` should not hold any value in between transactions. In practice, funds might be held by the contracts during a transaction and in case an attacker has control over the program execution he might call these functions to drain this funds during the transaction execution. We were not able to find any exploitable code path in the current version of the code, but this behavior clearly increases the surface area for potential attacks in potential future versions of the code.

It is also important to mention that the `PeripheryPayments` contract was originally out of scope for this review but in practice its implementation affects the derived contracts that are in scope.

Recommendation: Consider adding a non-reentrant guard to the `PeripheryPayments` which will be activated both on the 3 functions mentioned above and also on any potential function in the derived contracts.

Velodrome: We applied the recommended fix in commit 140a9fe5: inherit OZ's `ReentrancyGuard` in `PeripheryPayments` and add `nonreentrant` modifier to the above mentioned functions.

Spearbit: Verified.

5.3.13 `_teamEmissions` are using the decayed weekly rate instead of this week emissions

Severity: Low Risk

Context: Minter.sol#L145-L146

Description: In Minter.sol#L145-L146

```
uint256 _teamEmissions = (_rate * (_growth + _weekly)) / (MAX_BPS - _rate); /// @audit `_weekly` is the
↪  new week and not the prev one | _emission is prev
```

team emissions are a percentage of this weeks emission. The current week emission is calculated as (see Minter.sol#L147):

```
uint256 _required = _growth + _emission
```

Meaning that this week's mint is comprised of `_growth` and `_emission`. `_teamEmissions` is instead using `_weekly` which is decayed by an additional week, causing a loss to the team.

It's also worth considering it the divisor should be `(MAX_BPS - _rate);` or simply `MAX_BPS` as that's inflating the team emissions, making the `_rate` parameter harder to understand at first glance.

Recommendation: Rewrite the `_teamEmissions` math to:

```
uint256 _teamEmissions = (_rate * (_growth + _emission)) / (MAX_BPS);
```

Velodrome: We have applied the recommended fix in commit 4bf528a4 by rewriting the `_teamEmissions` math to the expression above.

Spearbit: Verified.

5.3.14 `VeloGovernor` Quorum is set to 25% which may make most proposals never reach quorum

Severity: Low Risk

Context: VeloGovernor.sol#L44-L50

Description: Governance in DeFi tends to have fairly low turnouts, a 25% quorum may make passing governance proposals close to impossible. For example:

- AAVE: 3%.
- COMP: 5%.
- UNI: 5%.

Overall, the setting would ensure that only the most heated proposals would ever reach quorum.

Recommendation: Consider if quorum should be that high based on past turnout.

Velodrome: At this stage, we will maintain the current quorum of 25% and can revise the value down in the future if needed. The protocols mentioned above do not have similar mechanics to escrow based tokens.

Protocols that offer a fairer comparison are the following:

- Curve: 15% (parameter proposals) and 30% (gauge proposals).
- Balancer: 25% (2M out of 7.9M veBal).
- Kyber: 20%.

Overall the above protocols include Convex and Aura, both having a ~48% share giving them an instant pass on any of the proposals.

The Velodrome Foundation has a 20% share, meaning that a 25% quorum makes governance more fair, leaving enough room for the community to not let the Foundation instantly pass a proposal (unlikely but possible in the context of Convex and Aura).

The team will be monitoring the effect of a larger quorum requirement and will consider a governor redeployment with a new quorum. Alternatively a proposal to lower the quorum parameters can also be considered.

Spearbit: Acknowledged.

5.3.15 `CLPool.initialize` implementation initializer could be re-initialized

Severity: Low Risk

Context: CLPool.sol#L139-L149

Description: Pools are deployed as proxies via the OZ `Clones` library, and their associated implementations require an initializer as replacement to a constructor. Just as a constructor, this should only be callable once, ideally on deployment.

The implementation currently relies on checking that the `factory` state variable is null to be initializable. It is set via the `_factory` param in the initializer, and there is no non-zero check on it. If a null `_factory` param were to be passed, it would allow for post-deployment initialization by anyone.

Recommendation: Consider having a non-zero check or omitting the `_factory` param and setting `factory` to msg.sender, which should always be the case with the current contract architecture.

Velodrome: The recommended fix has been applied in commit 140a9fe5: having a non-zero check on the `_factory` param.

Spearbit: Verified.

5.3.16 `SafeERC20.safeApprove()` is deprecated

Severity: Low Risk

Context: CLGauge.sol#L375, CLGauge.sol#L382

Description: The `SafeERC20.safeApprove` function from the OpenZeppelin's contracts used by the project is deprecated (SafeERC20.sol#L30-L37):

```
/**
 * @dev Deprecated. This function has issues similar to the ones found in
 * {IERC20-approve}, and its usage is discouraged.
 *
 * Whenever possible, use {safeIncreaseAllowance} and
 * {safeDecreaseAllowance} instead.
 */
function safeApprove(IERC20 token, address spender, uint256 value) internal {
    // ...
```

Recommendation: As recommended by OpenZeppelin, consider using `SafeERC20.safeIncreaseAllowance()`.

Velodrome: The recommended fix (replacing `safeApprove` with `safeIncreaseAllowance`) has been applied in commit 140a9fe5d.

Spearbit: Fixed as recommended in commit 140a9fe5.

5.3.17 `CLFactory.enableTickSpacing()` allows setting a 0 fee

Severity: Low Risk

Context: CLFactory.sol#L167

Description: Due to insufficient input validation, the `CLFactory.enableTickSpacing()` function lets the owner set 0 fee to a tick spacing. If this happens, the tick spacing is pushed into the `_tickSpacings` array, but the tick spacing won't still be active because fees must be positive (CLFactory.sol#L80). When `CLFactory.enableTickSpacing()` is called again to set a correct fee for the tick spacing, the tick spacing will be pushed into the array again, causing duplicates in the array.

Recommendation: In the `CLFactory.enableTickSpacing()` function, consider reverting if the value of `fee` is 0.

Velodrome: The recommended fix has been applied in commit 4baded78, and now the `fee` to be set in the `enableTickSpacing` function is required to be greater than 0.

Spearbit: Fixed as recommended in commit 4baded78.

5.3.18 Staked liquidity can be increased in a killed gauge

Severity: Low Risk

Context: CLGauge.sol#L213-L221

Description: `CLGauge.increaseStakedLiquidity()` allows to increase liquidity in a killed gauge. Unlike `CLGauge.deposit()`, it doesn't check if the gauge is active.

This allows to bypass the gauge liveliness check in `CLGauge.deposit()`:

1. A user stakes liquidity in an alive gauge via `CLGauge.deposit()`.
2. The user removes all their staked liquidity via `CLGauge.decreaseStakedLiquidity()`.
3. The gauge gets killed after some time.
4. The user stakes liquidity in the killed gauge by calling `CLGauge.increaseStakedLiquidity()`.

Recommendation: In `CLGauge.increaseStakedLiquidity()`, consider reverting if the value of `voter.isAlive(address(this))` is `false`.

Velodrome: The recommended fix has been applied in commit df5e6107 and now `CLGauge.inscreaseStakedLiquidity()` reverts if called on a gauge that is not alive.

Spearbit: Fixed as recommended in commit df5e6107.

5.4 Gas Optimization

5.4.1 Unnecessary zero-value initialization on state variables

Severity: Gas Optimization

Context: Pool.sol#L61-L64

Description: The two state variables are re-initialized to 0. This introduces unnecessary overhead with solidity on the deployment step, which raises gas ~2000 units per unnecessary assignment to 0.

Recommendation: Declared variables start with initial default values. In this case, they are 0, and it is needless to additionally assign them to 0 again, and will save a total of ~4000 gas on deploy:

```
- uint256 public index0 = 0;
- uint256 public index1 = 0;
+ uint256 public index0;
+ uint256 public index1;
```

Velodrome: The recommended fix has been applied in commit 34af3b94and now the variables are no longer being initialized.

Spearbit: Verified that commit 34af3b94implements the recommendation.

5.4.2 `Pool` unnecessarily uses fallback `Context._msgSender()`

Severity: Gas Optimization

Context: Pool.sol#L362

Description: The `Pool` contract appears to unnecessarily be using `_msgSender()` as it is using it from OZ's fallback `Context` contract without any intermediary meta transaction libraries between it. Thereby, it is just falling back to `msg.sender` but with unnecessary gas overhead.

It only makes sense to keep this if `Pool` may be intended to be an import of another contract that will support meta transactions, or if it's planned to be eventually implemented.

Recommendation: Replace instances of `_msgSender()` or `_msgData()` with their more direct and cheaper `msg.sender` and `msg.data` builtins, in the case of metatx not being actually utilized by the contract and any inheriting contract of it.

Velodrome: The recommended fix has been applied in commit 34af3b94and instances of '_msgSender' and '_msgData' have been replaced with 'msg.sender' and 'msg.data'.

Spearbit: Verified that commit 34af3b94implements the recommendation.

5.4.3 `for` loop increments could be unchecked

Severity: Gas Optimization

Context: Router.sol#L110-L116, Router.sol#L336-L351, Router.sol#L427-L445

Description: `Solc 0.8` introduces checked arithmetic by default. A number of contracts using this version have `for` loops, that could use `unchecked` blocks for the incrementing variable, with the potential to save a fair amount in gas costs for longer running loops.

Recommendation: Introduce `unchecked` blocks at the end of the loop for sufficiently simple incrementing loops, to save gas by foregoing overflow checks on the increment step. The introduction of more complex control flow with break or continue statements would require more care and possibly multiple `unchecked` blocks.

See Router.sol#L110-L116 for example:

```diff
- for (uint256 i = 0; i < _length; i++) {
+ for (uint256 i = 0; i < _length;) {
      address factory = routes[i].factory == address(0) ? defaultFactory : routes[i].factory; //
↪  default to v2
      address pool = poolFor(routes[i].from, routes[i].to, routes[i].stable, factory);
      if (IPoolFactory(factory).isPool(pool)) {
          amounts[i + 1] = IPool(pool).getAmountOut(amounts[i], routes[i].from);
      }
+     unchecked { ++i; }
  }
```

The increment being switched from `i++` to `++i` also saves gas, in the case of the compiler optimizer being off.

Similar changes can be applied across `0.8.x` contracts using `for` loops to save gas that meet the aforementioned conditions, only a subset within `Router.sol` have been noted for the sake of brevity.

Velodrome: The recommendation has been acknowledged, but we will not make any changes here.

Spearbit: Acknowledged.

5.4.4 The `forwarder` state variable is assigned to but unused in `CLGauge`

Severity: Gas Optimization

Context: CLGauge.sol#L84

Description: `forwarder` is assigned with value during the initialization of the contract but its value is never used and thus should be removed.

Velodrome: The recommended fix has been applied in commit df5e6107, and the unnecessary `forwarder` variable has been removed.

Spearbit: Verified that commit df5e6107 fix the issue by implementing the recommendation.

5.4.5 `_remaining` is not needed and `timeUntilNext` can be used instead in `CLGauge._notifyRewardAmount`

Severity: Gas Optimization

Context: CLGauge.sol#L346, Gauge.sol#L207

Description: The value of `timeUntilNext` and `_remaining` is identical in all scenarios. `timeUntilNext` is being used either way in the beginning of the function and thus `_remaining` can be replaced with `timeUntilNext`.

Velodrome: The recommended fix has been applied in commit df5e6107 by replacing the unnecessary variable `_remaining` with `timeUntilNext`.

Spearbit: Fixed in commit df5e6107 by implementing the recommendation.

5.4.6 Gas optimizations within `CLPool.collectFees`

Severity: Gas Optimization

Context: CLPool.sol#L949-L964

Description:

1. `collectFees` performs two `SLOAD` operations for the slot that stores the `gauge`, first as part of the `onlyGauge` modifier and second while emitting the `CollectFees` event. The latter can be saved by replacing `gauge` with `msg.sender`.
2. It performs arithmetic that aims to always set `gaugeFees.tokenX` in the branch to 1. The arithmetic can be simplified by setting `gaugeFees.tokenX` to 1, rather than performing some arithmetic prior to it, and passing the necessary decrement to `amountX` into the `TransferHelper` call by way of `TransferHelper.safeTransfer(token0, gauge, --amount0)` in the case of the `amount0` branch. These will reduce redundant arithmetic ops and caching.

Veoldrome: The recommended fix has been applied in commit 41d0e9af by simplifying the arithmetic operations and replacing `gauge` references with `msg.sender` to reduce `SLOAD` ops.

Spearbit: Fixed in commit 41d0e9af by implementing the recommendation.

5.4.7 `CLPool` state variables can be better tightly packed for gas efficiency

Severity: Gas Optimization

Context: CLPool.sol#L51-L54, CLPool.sol#L97-L105

Description: The `CLPool` contract declares a number of smaller type variables that could be packed into shared storage slots. Currently, their alignment appears supoptimal for gas efficiency purposes, meaning there are unnecessary `SLOAD` and `SSTORE` ops occurring.

Recommendation: The following recommendation is expected to be one of the more optimal alignments of state variables, such that they are packed into 2 storage slots, and in such a way that would decrease store operations across a number of functions:

```
uint256 public override periodFinish; // declare after any full size type

uint128 public override stakedLiquidity; // first slot
uint32 public override lastUpdated;
uint32 public override timeNoStakedLiquidity;
int24 public override tickSpacing;

uint128 public override liquidity; // second slot
uint128 public override maxLiquidityPerTick;
```

Note: `inheritdoc` comments were omitted just for clarity sake to focus on their recommended alignment.

Velodrome: The recommended fix has been applied in commit 41d0e9af by reordering the storage variables as recommended for better packing.

Spearbit: Verified that the recommendation has been applied in commit 41d0e9af.

5.4.8 `CLPool.swap` gas savings by not repeating no-op operations

Severity: Gas Optimization

Context: CLPool.sol#L601

Description: `CLPool.swap` performs in-memory operations. On each swap step, it calls the `fee` function, which is idempotent and as such its result can be cached to save around 400 gas per swap step.

Similarly `CLPool.swap` calls `_updateRewardsGrowthGlobal` each time it crosses a tick. The function will accrue past rewards only when `timeDelta != 0` meaning it's not necessary to call it after the first tick cross. Omitting the call should save at least 100 gas for each tick crossed when checking the timestamp for last update.

Recommendation: Consider adding:

- `fee` to `SwapState` and cache it.
- A `hasUpdatedFees` memory varaible to `SwapState` as to avoid re-calling the function.

Velodrome: The recommended fix has been applied in commit 41d0e9af by adding `fee` and `hasUpdatedFees` variables to `SwapState` to reduce unnecessary calls to their functions.

Spearbit: Verified. Fixed by extending `SwapState`.

5.4.9 `CLFactory` `SLOAD` gas savings

Severity: Gas Optimization

Context: CLFactory.sol#L104-L109

Description: The old `owner` in CLFactory.sol#L104-L109 is a storage variable and is read twice, you can save gas by caching it. This also applies to:

- `setSwapFeeManager`.
- `setUnstakedFeeManager`.

Recommendation: Consider applying the following hange:

```
  function setOwner(address _owner) external override {
+     address cachedOwner = owner;
+     require(msg.sender == cachedOwner); /// @audit cached
-     require(msg.sender == owner);
      require(_owner != address(0));
+     emit OwnerChanged(cachedOwner, _owner); /// @audit cached
-     emit OwnerChanged(owner, _owner);
      owner = _owner;
  }
```

Velodrome: Fixed in commit 140a9fe5.

Spearbit: Verified.

5.5 Informational

5.5.1 `DURATION` should be replaced with `VelodromeTimeLibrary.WEEK` to minimize the risk of a future error in `CLGauge`

Severity: Informational

Context: CLGauge.sol#L337

Description: In the current version of the code, `DURATION` and `WEEK` both hold the same value of `7 days` and are being used interchangeably in `CLGauge` and `VelodromeTimeLibrary`. However, CLGauge.sol#L337 may underflow in case `DURATION < timeUntilNext` which may happen in future versions of the code if `WEEK` holds a value greater than `7 days`. This underflow will totally corrupt the rewards accounting logic in `CLGauge`.

Recommendation: Consider removing the `DURATION` variable from `CLGauge` and instead use `VelodromeTimeLibrary.WEEK`.

Velodrome: The recommended fix has been applied in commit df5e6107 and the variable `DURATION` has been removed and replaced with the usage of `VelodromeTimeLibrary.WEEK`.

Spearbit: Fixed in df5e6107 by implementing the auditor's recommendation.

5.5.2 `CLGauge.updateRewards` function name does not follow the convention of _ prefix for internal functions

Severity: Informational

Context: CLGauge.sol#L96

Description: The repo implements the convention of adding a _ prefix to internal functions, but this is not the case with `CLGauge.updateRewards` and should be fixed.

Velodrome: The recommended fix has been applied in commit df5e6107 by renaming the internal function in question.

Spearbit: Fixed in df5e6107 by implementing the recommendation.

5.5.3 An operator can invalidate permits on behalf of the owner

Severity: Informational

Context: NonfungiblePositionManager.sol#L421-L423

Description: This finding is more of a gotcha than a real risk, as a malicious operator can just steal the tokens. This could be useful in whitehat attempts.

In the case of a nonce being signed by an owner, the operator could:

- Transfer to self (becomes owner of token).
- Sign a new permit.
- Use the permit to raise the nonce.

As a way to deny usage of nonces for the user (see NonfungiblePositionManager.sol#L421-L423):

```
function _getAndIncrementNonce(uint256 tokenId) internal override returns (uint256) {
    return uint256(_positions[tokenId].nonce++);
}
```

Recommendation: This may be useful to invalidate nonce in case of approval farming.

Velodrome: Acknowledged, no further action will be taken.

Spearbit: Acknowledged.

5.5.4 Nitpick: `Dispatcher` comment uses `bytes` instead of `Route`

Severity: Informational

Context: Dispatcher.sol#L132

Description: In the following instances:

- Dispatcher.sol#L132
- Dispatcher.sol#L143

`Dispatcher` states:

```
// equivalent: abi.decode(inputs, (address, uint256, uint256, bytes, bool))
```

But it actually decodes `Route[]` instead of `bytes`

Recommendation: Consider updating the comment.

Velodrome: The recommended fix has been applied in commit d90976a2 by updating the comment in question.

Spearbit: Verified.

5.5.5 Voting is using a snapshot in the future

Severity: Informational

Context: VetoGovernor.sol#L287-L297

Description: The Governors `propose` function stores a proposal in the following way (see VetoGovernor.sol#L287-L297):

```
_proposals[proposalId] = ProposalCore({
    voteStart: snapshot.toUint64(),
    proposer: proposer,
    __gap_unused0: 0,
    voteEnd: deadline.toUint64(),
    __gap_unused1: 0,
    executed: false,
    canceled: false,
    vetoed: false
});
```

The `snapshot` value is computed as (see VetoGovernor.sol#L284-L285):

```
uint256 snapshot = currentTimepoint + votingDelay();
```

Which is used as follows (see GovernorSimple.sol#L558):

```
uint256 weight = _getVotes(account, tokenId, proposal.voteStart, params);
```

Which will use the `voteStart` as the `timestamp` to look into, which is a time in the future.

While no specific attack has been found, it's worth keeping in mind that this can allow various entities to change their:

- Delegation.
- Spot amounts.
- Locks.

Which could result in drastic changes in voting power.

Recommendation: I believe it's worth monitoring for sudden changes in voting power, changing to using the time of the proposal could be gamed by the proposer, so there's no specific mitigation recommended beside monitoring participants behaviour.

Velodrome: The team is aware of this and made the change to give users more time to organize around proposals that matter to them.

Spearbit: Acknowledged.

5.5.6 `LateQuorum` is possible you will have to always veto malicious proposals

Severity: Informational

Context: VetoGovernorVotesQuorumFraction.sol#L13-L14

Description: As described in these resources (1, 2): the governor requires that quorum is met before the deadline, this means that a last second quorum may be reached for malicious proposals.

Recommendation: This implies that you must veto any malicious proposal, not vetoing may cause the proposal to pass at the last second. You can also look into this, which may help prevent this exact scenario.

Velodrome: The issue has been fixed in commit 5038ac35 integrating the OpenZeppelin late quorum module has been included. We will monitor for malicious proposals, and will scrutinize proposals that trigger the `ProposalExtended` event.

Spearbit: Verified.

5.5.7 `CLFactory.enableTickSpacing` is public but unused internally, could be used in constructor

Severity: Informational

Context: CLFactory.sol#L167, CLFactory.sol#L54-L68

Description: The `enableTickSpacing` function has a public visiblity specifier, but it is currently unused internally.

Recommendation: Consider replacing the lines on CLFactory.sol#L54-L68 to be calls to the function which would warrant its public specifier and also introduce its safeguards with much fewer LOCs and improved readability in sacrifice for some increased gas cost on the constructor.

Otherwise, set it to `external` only if it remains unused internally.

Velodrome: The recommended fix has been applied in commit 4baded78 and now `enableTickSpacing()` is also being used internally in the `CLFactory` constructor.

Spearbit: Verified that commit 4baded78 implements the recommendation.

5.5.8 `UniversalRouter.Permit2Payments` has unused imports

Severity: Informational

Context: Permit2Payments

Description: In Permit2Payments.sol#L6-L7

```
import {Constants} from '../libraries/Constants.sol';
import {RouterImmutables} from '../base/RouterImmutables.sol';
```

are unused imports

Recommendation: Consider removing them.

Velodrome: The recommended fix has been applied in d90976a2 by removing the unused imports.

Spearbit: Verified.

5.5.9 `UniversalRouter/Dispatcher` placeholder comment is wrong

Severity: Informational

Context: Dispatcher.sol#L344

Description: The comment states

```
// placeholder area for commands 0x22-0x3f
```

However, `0x22` is used for `APPROVE_ERC20`.

Recommendation: Change the comment to

```
// placeholder area for commands 0x23-0x3f
```

Velodrome: The recommended fix has been applied in commit d90976a2 by updating the command code in the comment.

Spearbit: Verified.

5.5.10 Usage of duplicating libraries

Severity: Informational

Context: CLGauge.sol#L7, CLGauge.sol#L20

Description: The `CLGauge` contract imports and uses `SafeERC20` and `TransferHelper` libraries, which implement identical functionality: safe transferring and approval of ERC20 tokens.

Recommendation: To keep the code cleaner and to avoid code duplication, consider using only `SafeERC20`.

Velodrome: The recommended fix has be applied in commit 6949c16d (i.e. use `SafeERC20`).

Spearbit: Fixed as recommended in commit 6949c16d.