PUBLIC

Code Assessment

of the Slipstream Dynamic Fee Smart Contracts

January 13, 2025

Produced for

by

Contents

1 Executive Summary

Dear Velodrome team,

Thank you for trusting us to help Velodrome with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Slipstream Dynamic Fee according to Scope to support you in forming an opinion on their security risks.

Velodrome implements the Slipstream Dynamic Fee module. A fee module that integrates with the Slipstream Concentrated Liquidity AMMs to provide the pools with dynamic fees which depend on market volatility.

The most critical subjects covered in our audit are soundness of the design, safety of arithmetics, and integration with concentrated liquidity oracles. Security regarding all aforementioned subjects is high.

In summary, we find that the codebase has a high level of security.

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

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	3
• Code Corrected	3

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 Slipstream Dynamic Fee repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	24 November 2024	797c70acd9989621f6f7566132c7e0bd03f94d4c	Initial Version
2	12 December 2024	479f512de8c72b863168d5b46788d21618d4e59f	After Intermediate Report

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

The following files are considered in scope for this assessment:

```
contracts/core/fees/CustomSwapFeeModule.sol
contracts/core/fees/CustomUnstakedFeeModule.sol
contracts/core/fees/DynamicSwapFeeModule.sol
```

2.1.1 Excluded from scope

Any contracts that are not explicitly listed above are out of the scope of this review. Namely, third-party libraries are explicitly out of the scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Velodrome offers a dynamic fee module for Velodrome Slipstream.

Velodrome Slipstream implements Concentrated Liquidity pools, which are a fork of Uniswap V3, implementing extra logic for rewards distribution for staked liquidity and custom fees. The current audit focuses on the integration of custom fees into the concentrated liquidity pool, and in particular reviews a new dynamic fee module which sets the swap and flashloan fees of the CL pool depending on the current market conditions.

The fee rate for swaps (and flashloans) is represented in the Slipstream Concentrated Liquidity pool (*CLPool*) as a part-per-million amount. This amount represents the portion of the swap's input amount that is payed as fee. Slipstream Concentrated Liquidity pools (*CLPool*) obtain the fee rate from the Concentrated Liquidity pool factory (*CLFactory*). In the *CLFactory*, the `swapFeeManager` can set the

`swapFeeModule`, which is a contract that implements the `getFee(address pool)` method. `getFee()` takes a pool address as input, and returns the current fee for that pool. In *CLFactory*, a stipend of 200k gas is set when calling the `swapFeeModule.getFee()`, and the fee returned is capped to 10%. In case the `swapFeeModule` is not set, or the call to the `swapFeeModule` reverts, the default fee for the pool's tick spacing is returned, which is configured in the `tickSpacingToFee` mapping of *CLFactory*.

Velodrome implements two Swap fee modules:

- *CustomSwapFeeModule*: allows the `swapFeeManager` to set a custom fee per pool.
- *DynamicSwapFeeModule*: returns a fee for the queried pool that depends on the observed volatility in the pool.

2.2.1 CustomSwapFeeModule

The contract *CustomSwapFeeModule* is a simple contract that allows the `swapFeeManager` defined in *CLFactory* to set custom fees for specific pools, through the `setCustomFee()` method. The fee has to be in a range of 0 to 3%. To set a zero fee, the sentinel value `420` has to be passed. The reason is that a value of `0` in the `customFee` mapping represents an unset custom fee for a pool. If a pool doesn't have a custom fee set, the default value in the `tickSpacingToFee` mapping of *CLFactory* is used.

The custom fees are queried with the `getFee(pool)` method.

2.2.2 DynamicSwapFeeModule

The *DynamicSwapFeeModule* contract implements a `getFee(pool)` method that returns a fee that is dependent on the volatility that is observed in the specified pool. The formula used in **Version 1** of the code (then revised in **Version 2**: Changes in Version 2), for a given pool, is:

$$f_T = \min(f_b + K|1 - \frac{i}{i_{twap}}|, f_C)$$

where f_T is the total fee, *f_b* is the base fee, that is the constant part of the fee that is independent of the volatility, K is a scaling factor that makes the dynamic fee more or less sensitive to volatility, `i` is the current tick, i_{twap} is the time-weighted average tick. f_C is the fee cap, the maximum fee that *DynamicSwapFeeModule* will return for a given pool.

The time-weighted average is evaluated over a time frame of length `secondsAgo`, which is a global variable shared by all pools. To know the time-weighted average tick of a pool, the `observe()` method is used to obtain snapshots of the "cumulative" tick at `secondsAgo` seconds in the past, and at the current time. The logic is the same as with Uniswap V3 oracles. The value returned by `observe([t])` is an integral of `tick * dt` at a given time t since the first operation in the pool. To obtain the average `tick` value between times `t1` and `t2`, `observe([t1, t2])` returns the values `(a1, a2)`. We can take the difference and divide by the time delta to obtain the average tick: `(a2 - a1)/(t2 - t1)`.

Parameters f_b (`baseFee`), K (`scalingFactor`), f_C (`feeCap`) are configurable on a per pool basis. If `baseFee` is not defined, the default value returned by `tickSpacingToFee()` of *CLFactory* is used. If `scalingFactor` is not set, the default values `defaultScalingFactor` and `defaultFeeCap` are used.

The parameters can be set by the `swapFeeManager`, defined in *CLFactory*. `baseFee` (per pool) has to be less than `3e4` (3%), and a value of 420 is a sentinel value for a zero base fee (`0` being reserved for unset `baseFee`, i.e. defaulting to `tickSpacingToFee`). `feeCap` (per pool) and `defaultFeeCap` (global) have to be smaller than 5%. `scalingFactor` (per pool) and `defaultScalingFactor` have to be smaller than `1e18`, and `scalingFactor` for a pool can only be set if `feeCap` has already been configured.

The following setters are available to `swapFeeManager`:

- `setCustomFee()`: sets the `baseFee` for a pool.
- `bulkUpdateFees()`: allows setting the `baseFee` for multiple pools at once.

- `setDefaultFeeCap()`, `setDefaultScalingFactor()`: set the defaults.
- `setFeeCap()`: sets the `feeCap` for a pool.
- `setScalingFactor()`: sets the `scalingFactor` for a pool.
- `resetDynamicFee()`: clears the dynamic fee settings (`feeCap` and `scalingFactor`) of a pool.
- `setSecondsAgo()`: sets the global lookback time of ticks time-weighted average.

When setting the dynamic fee parameters for a given pool, three calls must be performed:

1. `setCustomFee()`
2. `setFeeCap()`
3. `setScalingFactor()`

Velodrome communicated that after deployment the `defaultScalingFactor` will be `0` and the `defaultFeeCap` will be `50_000` such that only the base fee is used for a pool that has not been configured with a dynamic fee. The base fee for a pool can be set in the constructor of the *DynamicSwapFeeModule* contract or with `setCustomeFee()`.

The `swapFeeManager` can `registerDiscounted()` addresses, which will benefit from a discount on the fees. The discount value is set per address, and can be set to a value of up to 50%. `deregisterDiscounted()` removes the discount for an address and is only callable by the `swapFeeManager`.

2.2.3 CustomUnstakedFeeModule

The fees levied by a pool are distributed to Liquidity Providers and voters of the pool's gauge in the Velodrome voting system. Liquidity providers that choose to stake their LP tokens in the pool's gauge forfeit their portion of the fees, and those go to the gauge voters instead. LP providers that do not stake their LP tokens receive the corresponding fees. However, a portion of the fees generated by the liquidity of LP providers who do not stake (unstaked liquidity) is anyway directed to the gauge voters. This "tax", aimed to incentivize staking, is called `unstakedFee()` in the *CLPool* codebase, and is queried by the pool by calling the `getUnstakedFee(address pool)` method of the *CLFactory*. The factory in turn queries the `unstakedFeeModule` through its `getFee(address pool)` method. The `unstakedFeeModule` address can be configured by the `unstakedFeeManager` role. The factory accepts a value of up to 100% for the unstaked fee, and otherwise defaults to `defaultUnstakedFee`, which is configurable by the `unstakedFeeManager` to a value of less than 50%.

The contract `CustomUnstakedFeeModule` implements the `getFee()` method and allows the `unstakedFeeManager` to set pool specific custom unstaked fees. Pools which are not configured default to an unstaked fee of 10%, and the maximum value accepted by `setCustomFee()` is 50%. A fee of 0 can be configured by setting the sentinel value `420`, as `0` is reserved to signal a non-configured pool.

2.2.4 Changes in Version 2

In Version 2, the dynamic fee module has been updated to use a new formula for the fee calculation. The new formula is:

$$f_T = \min(f_b + K|i - i_{twap}|, f_C)$$

where f_T is the total fee, f_b is the base fee, K is a scaling factor, `i` is the current tick, i_{twap} is the time-weighted average tick. f_C is the fee cap, the maximum fee that *DynamicSwapFeeModule* will return for a given pool.

3 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.

4 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.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Dynamic Fee Depends on the Price Value Code Corrected

Low-Severity Findings	3

- Dynamic Fee Is Ignored When Tick Is 0 Code Corrected
- Missing Checks in Constructor Code Corrected
- Missing Event Emission in Constructor Code Corrected

6.1 Dynamic Fee Depends on the Price Value

Correctness Medium Version 1 Code Corrected

CS-VELODYNFEE-001

The dynamic fee computation in `DynamicSwapFeeModule` is meant to depend on the price volatility. From any price p, an increase of 0.01% in the price should result in the same increase in the fee. However, the implementation in Version 1 does not reflect this behavior as the dynamic fee depends on the current price value.

For example, let's consider the following scenario:

1. Case 1:

We have the following values :

- `twAvgTick = 1`
- `currentTick = 2`

The dynamic fee is computed as follows:

$|K \cdot (1 - \frac{2}{1})| = = K$

1. Case 2

We have the following values :

- `twAvgTick = 69081`
- `currentTick = 69082`

This corresponds to a price of 1000.

The dynamic fee is computed as follows:

$|K \cdot (1 - \frac{69082}{69081})| \approx 0$

In the first case, the dynamic fee is equal to K while in the second case, the dynamic fee is equal to 0 while the increase in price is 1 tick (0.01%) in both cases.

Code corrected:

The formula has been modified to use the difference in ticks, such that the fee depends only on the change of price (difference in logarithms), and not on the magnitude of the price.

The new formula is:

$f = K|tick - TWAVG(ticks)|$

6.2 Dynamic Fee Is Ignored When Tick Is 0

Correctness | Low | Version 1 | Code Corrected

CS-VELODYNFEE-003

Due to the nature of the dynamic fee computation, the `twAvgTick` value cannot be 0 as it would result in a division by zero. In this case a dynamic fee of 0 is returned. This is not correct as it does not account for the fact that the `currentTick` value could be larger or smaller than the `twAvgTick` value indicating volatility in the price. Therefore, pools where tokens are traded at a price close to 1 can have a dynamic fee of 0 even if the price is volatile.

Code corrected:

This issue has been fixed together with `CS-VELODYNFEE-001`, by modifying the formula for dynamic fee computation.

6.3 Missing Checks in Constructor

Correctness | Low | Version 1 | Code Corrected

CS-VELODYNFEE-004

In `DynamicSwapFeeModule`, the constructor does not verify that `defaultScalingFactor` is less or equal to the `MAX_SCALING_FACTOR` and that `defaultFeeCap` is less or equal to the `MAX_FEE_CAP`.

Code corrected:

Checks have been added to the constructor to ensure that the values of `defaultScalingFactor` and `defaultFeeCap` are within the expected bounds.

6.4 Missing Event Emission in Constructor

Correctness | Low | Version 1 | Code Corrected

CS-VELODYNFEE-005

In `DynamicSwapFeeModule`, the constructor does not emit `DefaultScalingFactorSet` and `DefaultFeeCapSet` events after setting the default scaling factor and default fee cap.

Code corrected:

Events are now emitted after setting the default scaling factor and default fee cap in the constructor of `DynamicSwapFeeModule`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Dynamic Fee Always Rounds Towards Zero

Informational **Version 1** **Acknowledged**

CS-VELODYNFEE-007

In `DynamicSwapFeeModule`, the dynamic fee computation computes the time-weighted average tick in the following way:

```
twAvgTick = int24((tickCumulatives[1] - tickCumulatives[0]) / _secondsAgo);
```

However, `tickCumulatives` are signed integers and the division operation will always round towards zero. Thus, if the difference between `tickCumulatives[1]` and `tickCumulatives[0]` is negative, the time-weighted average tick will be rounded up, towards zero while if the difference is positive, the time-weighted average tick will be rounded down, towards zero. Since ticks represent logarithms of prices, it amounts to rounding up if the price is below 1.0, and rounding down if the price is above 1.0.

The same applies to `(scaledK - scaledK * currentTick / twAvgTick) / PRECISION` with `currentTick` being a signed integer.

Risk accepted:

Velodrome acknowledges the behavior with the following statement:

> The quirk in division is noted but is not considered to significantly change the outcome of the fees calculated.

7.2 Dynamic Fee Is Always Computed

Informational **Version 1** **Acknowledged**

CS-VELODYNFEE-008

The specification state that :

`The default parameters are also mutable. It is possible to set up the dynamic fee module such that it is only using the base fee.`

While it is indeed possible to set up the dynamic fee module such that it is only using the base fee, this requires setting the `baseFee` parameters and `feeCap` to the same value, and the dynamic fee will still be computed every time from the oracle observations even though it is not used, resulting in useless gas consumption.

Risk Accepted:

Velodrome acknowledges the issue with the following statement :

> The gas usage by the function is low so this is accepted.

8 Notes

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

8.1 Dynamic Fee Is 0 While the Observation Cardinality Has Not Grown

Note Version 1

In `DynamicSwapFeeModule._getDynamicFee()`, the following check is performed:

```
if (observationCardinality < _secondsAgo / MIN_SECONDS_AGO) return 0;
```

This check ensures that the oracle's observation cardinality is large enough to contain observations from `_secondsAgo` in case observation writes are happening every block. If the cardinality is not large enough, the dynamic fee is 0.

When the pool is first created, the observation cardinality is 1. This means that the dynamic fee will be 0 until the observation cardinality has grown. However, the increase of cardinality of the ring buffer only happens when the oracle is writing an observation in the last slot of the ring buffer. This means that an update of the observation cardinality from `n` to `m` where `n < m` will be in the worst case only be reflected after `n` blocks. For example, in the typical case where `_secondsAgo == 3600` (1 hour), `observationCardinality` is required to be `1800`. If the cardinality was previously increased to `1799`, it is possible that `1799` observation writes are required before the dynamic fee becomes operative. This might take 1 hour in the best case, if observation are written every single block.

8.2 Dynamic Fee Updates After the Swap

Note Version 1

Dynamic fees only update after a swap. This means that, in volatile market conditions characterized by jumps in prices, the arbitrageurs will trade in the pool when the current tick is still close to the historical one, and they will be able to perform large trades which significantly move the current tick toward the new price while only paying for the base fee. In the extreme case where all the price evolution of a pool is characterized by rare price jumps, the dynamic fees will not mitigate the losses of the Liquidity Providers.