PUBLIC

Code Assessment

of the Superchain Slipstream Smart Contracts

13 November, 2024

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 Superchain Slipstream according to Scope to support you in forming an opinion on their security risks.

Velodrome implements Superchain Slipstream, a Superchain extension of the Velodrome Slipstream concentrated liquidity pools and liquidity mining incentives. Superchain Slipstream allows deploying Concentrated Liquidity pools and gauges on Leaf chains, chains which are part of the Optimism Superchain ecosystem. The Leaf chain gauges will receive rewards for Liquidity Providers in the form of Velodrome emissions bridged from the Root chain (OP Mainnet).

The most critical subjects covered in our audit are integration with the Velodrome superchain system, cross-chain compatibility, and gauge liquidity accounting. The security of all aforementioned subjects is high, after some of the issues uncovered by ChainSecurity were properly addressed.

The general subjects covered in our audit are ABI compatibility of similar contracts, address collisions, correct deployment of pools and gauges. The security of all aforementioned subjects is high. Possibility of address collisions are discussed in the note Attackers can in the future generate address collisions.

In summary, we find that the codebase provides 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	1
• Code Corrected	1
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	1
• Code Corrected	1

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 Superchain Slipstream 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	7 Oct 2024	c82627fa4542d937b6e4ff6914fb190ce1227e64	First version
2	30 Oct 2024	47ab65233005a70acb14f6673f2b6fd2c3ca3e2d	Fixes and Mode sfs
3	1 Nov 2024	80fb1682f6d1bfa8852a4cbed76286caff02f742	Third Version
4	5 Nov 2024	e404e3d8e80faf7b01bc2435b6ef508bfe22d4ee	Fourth Version

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

The assessment has been partially performed as a *diff* audit, and partially as a full audit. In the *diff audit* only the changes compared to the previous codebase have been evaluated. The reference commit used as the base of the difference is `55b677900751f89566fb19d72d091cc101a9d28b`, referred to as `BASE_VERSION`. The following files are part of the diff audit (arrows indicate renaming compared to the base version):

```
contracts/core/CLFactory.sol
contracts/core/CLPool.sol
contracts/gauge/CLGauge.sol -> contracts/gauge/LeafCLGauge.sol
contracts/gauge/CLGaugeFactory.sol -> contracts/gauge/LeafCLGaugeFactory.sol
contracts/gauge/libraries/SafeCast.sol
contracts/periphery/NonfungiblePositionManager.sol
```

The following files are new in Version 1, and have been fully audited.

```
contracts/mainnet/gauge/RootCLGauge.sol
contracts/mainnet/gauge/RootCLGaugeFactory.sol
contracts/mainnet/pool/RootCLPool.sol
contracts/mainnet/pool/RootCLPoolFactory.sol
```

For Version 1, the integrations with the overall Velodrome systems have been evaluated taking the following commits as references:

```
contracts-private              ee15bd1e63d3b33ce8d179f73bca7390812bd99b
superchain-contracts-private   45bfd414892adabd95103669345418d4080fb4bc
```

2.1.1 Scope after Version 2

With Version 2, the scope is expanded to include contracts specific to the Mode network. Contracts previously in scope have also been renamed. The scope after Version 2 consists of:

```
Leaf:
    contracts/core/CLFactory.sol
    contracts/core/CLPool.sol (only **diff-audited**)
    contracts/gauge/LeafCLGauge.sol (only **diff-audited**)
    contracts/gauge/LeafCLGaugeFactory.sol
    contracts/periphery/NonfungiblePositionManager.sol (only **diff-audited**)

Root:
    contracts/root/gauge/RootCLGauge.sol
    contracts/root/gauge/RootCLGaugeFactory.sol
    contracts/root/pool/RootCLPool.sol
    contracts/root/pool/RootCLPoolFactory.sol

Mode:
    contracts/extensions/ModeFeeSharing.sol
    contracts/core/extensions/ModeCLFactory.sol
    contracts/core/extensions/ModeCLPool.sol
    contracts/gauge/extensions/ModeLeafCLGauge.sol
    contracts/gauge/extensions/ModeLeafCLGaugeFactory.sol
    contracts/periphery/extensions/ModeNonfungiblePositionManager.sol
    contracts/periphery/extensions/ModeSwapRouter.sol
```

2.1.2 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.

In this report, we assume that the slipstream codebase at `BASE_VERSION` is safe. For contracts marked as **diff-audited**, only the changes between the `BASE_VERSION` and the latest commit have been audited.

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 implements an extension of the Uniswap V3 protocol to be used on Velodrome Superchain. By doing so, Superchain Slipstream offers access to Velodrome liquidity rewards to liquidity providers of Concentrated Liquidity pools on chains which are part of the Superchain.

In what follows, we look into different components of the system.

2.2.1 NFT Position Manager

Like in Uniswap V3, LPs can, through the `NonfungiblePositionManager` contract, deposit funds as liquidity in a pool and price range and receive an NFT as a receipt. Once holding an NFT, LPs can

1. increase their liquidity by depositing the required amount of token-0 & token-1
2. decrease their liquidity by burning the liquidity deposited in a position
3. collect the fees accumulated in their position
4. and burn their NFT, once their position is cleared (no owed amounts left)

Position NFTs, differently than positions held in the pool directly, are transferable.

2.2.2 Leaf-Chain Concentrated Liquidity Pool Factory

Through the `CLFactory` contract, users can deploy a pool defined by its token-0, token-1, tick spacing, and the initial price. This pool gets deployed as a minimal proxy through the *Openzeppelin Clones* library, with a deployment salt depending on: token pair and the tick spacing. Right after contract creation, the pool gets initialized with the given configuration. It is worth mentioning that for a tick spacing to be valid, and hence usable when deploying pools, it has to be enabled in the factory.

2.2.3 Leaf-Chain Concentrated Liquidity Pools

The `CLPool` contract implements Concentrated Liquidity (CL) pools on the Leaf Chains, which are a slightly modified version of Uniswap V3 pools. Similarly as in Uniswap V3, LPs can

1. deposit liquidity to a price range, defined by an upper and lower tick, by transferring the corresponding amount of token-0 and token-1 to the system.
2. collect fees earned by the liquidity in token-0 & token-1 from the pool.
3. burn a given amount of liquidity and receive token0 and token1 back.

On top of what is offered by Uniswap V3, it also implements *staking* functionality. The `stake()` function can only be called by the pool's gauge. This function updates the rewards, and then transfers liquidity from the position owned by the NFT to the gauge. Considering this fact, the overall liquidity of the pool would be divided to two parts:

1. staked liquidity: it represents the amount of liquidity deposited in the gauge. If this liquidity is used during swaps, the fees are accrued for the gauge. As Uniswap V3 math tracks every value (rather than price) per unit of liquidity, rewards are multiplied by this.
2. unstaked liquidity, which is the position's liquidity still held by the LPs. This value is used when calculating the fees in favor of LPs and updating owed values.

Those positions owned by the gauge also have accumulated fees which can be collected by the gauge.

2.2.4 Leaf-Chain Gauge

As mentioned before, LPs' positions on the Leaf Chain's Concentrated Liquidity Pool are represented by NFTs. LPs can then deposit their NFTs in the `LeafCLGauge`, which means relinquishing their right on collecting fees of CL pool and receive rewards (Velo emissions) in return.

When depositing an NFT in the gauge, first, all the accrued fees are collected and sent to the LP, then NFT is transferred from the position owner to the gauge, and finally the meta information regarding the two positions (both have the same lower and upper bounds but with different owners; one belongs to the NFT Manager and the other to the gauge) gets updated.

LPs, who have staked their NFTs in the gauge, can collect their rewards over each position held by them. Collecting the rewards first updates the rewards growth of the position in the pool, and then transfers the accrued amount of reward token to the receiver.

From the root chain, rewards are sent through the Velodrome bridge to the leaf gauges. Rewards are deposited through the `notifyRewardAmount()` function.

2.2.5 Leaf-Chain Gauge Factory

Voter contract can deploy a new gauge for a pool with its voting reward contract. The gauge gets deployed with create3 (similar to create2, but doesn't include the initialization code in the address calculation) with a salt depending on chain ID, token pair, and tick spacing of the pool.

On the Root Chain we have the following contracts:

2.2.6 Root Chain Liquidity Pool

The `RootCLPool` contract is a place holder that implements getter methods for the purpose of integrating with the existing Velodrome system on the Root chain.

2.2.7 Root Chain Pool Factory

The contract `RootCLPoolFactory` is a factory for creation of `RootCLPool`. Pools are deployed as minimal proxies through OpenZeppelin Clones library. They share the same implementation (`RootCLPool`) and use as deployment salt a hash of `chainid` , `token0`, `token1`, `tickSpacing`.

2.2.8 Root Chain Gauge

On `RootCLGauge`, the `Voter` can notify new rewards to the Gauge with `notifyRewardAmount()`. The reward amount is capped by the maximum weekly reward for this gauge (If extra tokens are sent, they will be transferred to the `Minter`). The rewards are then bridged to the leaf chain through a message of type `NOTIFY`. The notify admin of the gauge factory can also call `notifyRewardWithoutClaim()` which similarly create a message of `NOTIFY_WITHOUT_CLAIM` and sends it to the leaf chain to be processed.

2.2.9 Root Chain Gauge Factory

The `RootCLGaugeFactory` contract allows deployment of root-leaf chain gauge pairs. The `Voter` on the root chain calls `createGauge()`. First it deploys a gauge for a the given pool on the root chain. Then, it creates a message of type `CREATE_GAUGE` and sends it through the bridge to the leaf chain. The message is forwarded to the leaf voter, which calls `createGauge()` on the `LeafCLGaugeFactory`, which deploys the leaf gauge.

`RootCLGaugeFactory` also exposes the `calculateMaxEmissions()` function. This function, given a Gauge address, returns the maximum weekly reward amount for that gauge during the current week. Max emissions for a gauge are configured as a percentage of the total weekly reward emissions. An emission admin can set a specific emission cap for a gauge, through the `setEmissionCap()` function. If a gauge does not have a cap set, the global `defaultCap` is used instead.

2.2.10 Factory Deployment

The `RootCLGaugeFactory` and `RootCLPoolFactory` on OP Mainnet and the `LeafCLGaugeFactory` and `CLFactory` on each Leaf chains are deployed in a permissioned way through `CreateX.create3`, such that they have the same address on all chains.

2.2.11 Trust Model and Assumptions

`RootCLGaugeFactory.emissionAdmin` can set the emission cap for a gauge, as well as the default cap. These values are used when the Voter calls `notifyRewardAmount()` on the root chain. Hence, this role is trusted.

`RootCLGaugeFactory.notifyAdmin` can call `notifyRewardWithoutClaim()` and is trusted.

`RootCLPoolFactory.owner` enables the tick spacings and sets fees for them. It is trusted.

`CLFactory.owner` sets the fee modules. It is trusted.

The reward token is assumed to be the Velo V2 token.

Reward distribution to gauges (`notifyRewardAmount()`), by calling `Voter.distribute()`, is assumed to happen on a weekly basis.

2.2.12 Changes in Version 2

Velodrome has extended the functionality of the ecosystem to be able to participate in Sequencer Fee Sharing (SFS) program of Mode Network. It allows owners of smart contracts to receive a portion of fees when users interact with their smart contracts. `ModeFeeSharing` provides the functionality of registering in SFS right upon deployment. `ModeCLFactory` inherits from `ModeFeeSharing` and registers itself on SFS contract. An NFT is minted that allows claiming the fees originated in the contract. The `ModeCLPools` assign their sequencer fees to the already existing token of the factory (simplifying the fee collection). `SwapRouter` also links its fee to the token of the factory.

`ModeNFTPositionManager` inherits from `ModeFeeSharing`, and registers itself with SFS. `ModeLeafCLGauge` and `ModeLeafCLGaugeFactory` link their fees to the token of `ModeNFTPositionManager`.

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:

- Design: Architectural shortcomings and design inefficiencies
- 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	1

- increaseLiquidity() for NFT Owned by Gauge Breaks Accounting of Liquidity Code Corrected

High-Severity Findings	1

- RootCLPoolFactory Does Not Support Pools With Same Parameters on Different Chains Code Corrected

Medium-Severity Findings	2

- NonFungiblePositionManager Can Return Incorrect Lists of User Positions Code Corrected
- Gas Limit of createGauge on Leaf Chain Incompatible With Slipstream Code Corrected

Low-Severity Findings	1

- Different Interfaces in Slipstream Contracts Code Corrected

Informational Findings	1

- Emission Cap Not Sanitized Can Cause Overflow Code Corrected

6.1 `increaseLiquidity()` for NFT Owned by Gauge Breaks Accounting of Liquidity

Design Critical Version 3 Code Corrected

CS-VELOSLIP-008

In Version 3 a critical check is removed from `increaseLiquidity()`, allowing anybody to increase the liquidity of a position NFT owned by the gauge. This can lead to incorrect liquidity accounting in the pool and gauge, with potentially critical consequences.

The `increaseLiquidity()` function of `NonfungiblePositionManager` allows increasing the liquidity of a tokenised position. It is unpermissioned, such that any user can increase any other user's liquidity, however Version 1 had the following line, which prevented increasing the liquidity of tokenised positions owned by the gauge:

```
if (ownerOf(params.tokenId) == gauge) require(msg.sender == gauge);
```

The check has been removed in Version 3. The consequence is that the liquidity of a tokenised position owned by the Gauge, and the amount of liquidity the gauge has staked in the pool can be different, since no `pool.stake()` is triggered by `increaseLiquidity()`. This results in a tokenised position whose liquidity is only partially staked. When the position NFT is unstaked, the full liquidity amount is unstaked from the pool, even if only part of that liquidity had been staked for that NFT. This can take liquidity from the staked positions of other users, leaving them unable to withdraw. The incorrect accounting of liquidity on withdrawal can also result in the incorrect attribution of rewards.

The following example aims to expose a transaction sequence resulting in incorrect behavior:

- User Bob (B) mints nft 1, depositing 100 liquidity in range R.
- User Mallory (M) mints nft 2, depositing 100 liquidity in range R. From the perspective of the pool, the NonfungiblePositionManager (nfpm) has 200 liquidity in range R.
- B deposits nft 1 in the gauge. The gauge calls pool.stake(), which reduces the liquidity of the nfpm for range R, and increases the liquidity of the gauge for range R
- M deposits nft 2 in the gauge. From the pool perspective now the nfpm has 0 liquidity in range R, and the gauge has 200 liquidity.
- M calls increaseLiquidity() to increase liquidity for nft 2 by 100. Nft 2 belongs to the gauge, and has now 200 liquidity from the perspective of nfpm. The new liquidity added by increaseLiquidity() goes to the nfpm from the perspective of the pool, so the liquidity of nft 2 is split between 100 for the gauge and 100 for nfpm. From the pool perspective, the gauge still has 200 liquidity total, and nfpm 100
- M withdraws nft 2 from the gauge. This causes a call of pool.stake(-200), since the whole nft amount is unstaked, even if M actually only owns 100 staked liquidity (the other 100 liquidity is in the nfpm position). From the pool perspective, now the gauge has 0 liquidity, and nfpm 100 liquidity.
- B tries to withdraw nft 1 from the gauge. This triggers stake(-100), which fails because it underflows, since the gauge has 0 liquidity left in the pool. B is prevented from recovering their liquidity.

The consequence is that users can be locked from accessing their liquidity. The rewards for existing staked liquidity can also be stolen by an attacker.

Code corrected:

In Version 4, increaseLiquidity() cannot be called on gauge-owned tokenIds. The following assertion has been added:

```
require(ownerOf(params.tokenId) != pool.gauge());
```

6.2 RootCLPoolFactory Does Not Support Pools With Same Parameters on Different Chains

Design | High | Version 1 | Code Corrected

CS-VELOSLIP-001

In the method createPool() of RootCLPoolFactory, the following check ensures that the same pool is not deployed twice:

```
require(getPool[token0][token1][tickSpacing] == address(0), "AE");
```

However, the check is too strict, as it does not include the chainid of the new pool. It is reasonable that two pools with equal parameters may be deployed on different leaf chains, let us say one with chainid 11 and the second with chainid 12. This is possible within the context of Superchain Slipstream by using the createPool() function of the leaf pool factory. However, only one of the two pools can have a gauge deployed, since gauge deployment requires creating a *mock* RootCLPool on the root chain. When this mock pool is deployed, for example for the pool on chainid 11, its address is saved in the getPool mapping:

```
getPool[token0][token1][tickSpacing] = pool;
```

This mapping does not consider the pool's chainid. The pool with the same parameters but chainid 12 will therefore have to occupy the same slot in the mapping, preventing the deployment of the required RootCLPool.

Code corrected:

Velodrome has modified the storage variable `getPool` to embed the chainid as well:

```
mapping(uint256 => mapping(address => mapping(address => mapping(int24 => address)))) public override getPool;
```

6.3 NonFungiblePositionManager Can Return Incorrect Lists of User Positions

Correctness **Medium** **Version 2** **Code Corrected**

CS-VELOSLIP-009

Version 2 introduces a mapping from user and pool addresses to enumerable sets of positions, in `NonFungiblePositionManager`. This mapping allows quick enumeration of user positions for a given pool through the `userPositions(user, pool)` view function. This function can return incorrect results, as the mapping is updated when minting and burning position NFTs, but not when transferring them from one user to another.

The mapping `_userPositions`, defined as

```
mapping(address => mapping(address => IterableEnumerableSet.UintSet)) internal _userPositions;
```

keeps track of an enumerable set of tokenIds for every user and pool pair. When minting a new liquidity position in `NonFungiblePositionManager`, the newly minted `tokenId` is added to the mapping for the recipient as:

```
_userPositions[params.recipient][address(pool)].add(tokenId);
```

and when burning, the `tokenId` is removed from the mapping of the user and pool as:

```
_userPositions[ownerOf(tokenId)][pool].remove(tokenId);
```

However, the `tokenId` can be transferred from one user to another. When transferred, no change from one user's set to another is implemented, such that it will remain in the set of the minting user. This does not cause internal problems, since trying to remove an element which is not in an enumerable set will not revert. However, the `userPositions()` function might return incorrect results if the queried user has transferred their NFTs.

Code corrected:

`NonFungiblePositionManager` in Version 3 implements the `_beforeTokenTransfer()` hook of OpenZeppelin's `ERC721`. The hook is called before every token transfer, and ensures that the `tokenId` is removed from the sender's enumerable set and added into the receiver's enumerable set.

6.4 Gas Limit of createGauge on Leaf Chain Incompatible With Slipstream

Design | Medium | Version 1 | Code Corrected

CS-VELOSLIP-002

The file `GasLimits.sol` of the `superchain-contracts-private` repository specifies a limit of 5.8M gas to execute the `CREATE_GAUGE` command on the leaf chain. This limit is compatible with the gas consumption of `CREATE_GAUGE` for non *slipstream* gauges, however it is too low for *slipstream* gauges. Overall `CREATE_GAUGE` (which includes pool deployment, fees and bribes reward contracts deployment, and gauge deployment) requires around 6.1M gas for slipstream, because `LeafCLGauge` has a bigger bytecode size than `LeafGauge`. Deploying the system with current gas limit would result in a gauge creation message to be stuck in the bridge, until an operator manually replays it with more gas.

Code corrected:

Velodrome has increased the gas limit for `CREATE_GAUGE` command to `7_320_000` gas units.

6.5 Different Interfaces in Slipstream Contracts

Design | Low | Version 1 | Code Corrected

CS-VELOSLIP-003

Some contracts differ in their interfaces and exposed behaviors between their slipstream and their non-slipstream versions. This could cause issues for integrators.

View method `isPair(pool)` in *slipstream* `RootCLPoolFactory` returns true if a `RootCLPool` has been deployed at that address. In non-slipstream `RootPoolFactory`, `isPair()` always returns `false`. This also translates to a different behavior when calling `createGauge` on the `Voter`. Superchain Slipstream pools can have unpermissioned gauge deployment if both tokens are whitelisted, but Superchain non-slipstream pools cannot be deployed in an unpermissioned manner.

`RootPoolFactory` exposes the `isPool()` view, equivalent to `isPair()`. However, `RootCLPoolFactory` only has `isPair()` and not `isPool()`.

`RootPoolFactory` exposes the view methods `allPools()` and `allPools(uint256)`. `RootCLPoolFactory` only implements `allPools(uint256)`

`RootPoolFactory` exposes a `implementation()` getter. `RootCLPoolFactory` exposes a `poolImplementation()` getter.

`RootPool` exposes `chainid()`. `RootCLPool` exposes `chainId()` (different capitalization).

Code corrected:

`RootCLPoolFactory.isPair()` now always returns false. Furthermore, `isPool()` as well as `allPools()` functions have also been added. `RootPoolFactory.poolImplementation()` has been renamed to `implementation()`. Also, the public storage variable `RootCLPool.chainId` has been renamed to `chainid`.

6.6 Emission Cap Not Sanitized Can Cause Overflow

Informational | Version 1 | Code Corrected

CS-VELOSLIP-005

The `emissionAdmin` of `RootCLGaugeFactory` is trusted, and we assume they will not set malicious values as a gauge emission cap, through the functions `setEmissionCap()` and `setDefaultCap()`. However if a too high value is set, for example for the purpose of having no reward cap for a gauge, the multiplication `_weeklyEmissions * maxRate` in `calculateMaxEmissions()` can overflow. This can cause loss of rewards.

Code corrected:

In `RootCLGaugeFactory`, both functions `setEmissionCap()` and `setDefaultCap()` enforce `_emissionCap` and `_defaultCap` respectively not to exceed `MAX_BPS`.

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 Inconsistent Code in NonfungiblePositionManager

Informational | Version 1 | Code Partially Corrected

CS-VELOSLIP-006

Function `increaseLiquidity()` of `NonfungiblePositionManager` is no longer called by `LeafCLGauge`. However it contains the line:

```
if (ownerOf(params.tokenId) == gauge) require(msg.sender == gauge, "NG")
```

This line implies that the *gauge* could potentially call `increaseLiquidity()`, maybe in a future code iteration. However, if `increaseLiquidity()` was to be called with the gauge as recipient, the liquidity addition would be invalid, as the liquidity is added to the position of the `NonfungiblePositionManager` (`address(this)`), instead of the `gauge`.

The code is therefore misleading, pointing to possible uses which would actually be invalid. In practice, the gauge does not call `increaseLiquidity()`, and `increaseLiquidity()` should not be called with `tokenId` owned by the gauge.

Similarly, `decreaseLiquidity()` is now incompatible with calls from the gauge, which is fine since the gauge never calls it. However it would be clearer to explicitly disallow that possibility.

Explicitly disallowing the gauge to call these functions would result in clearer code.

In Version 4 liquidity increasing for gauge positions is explicitly forbidden. `decreaseLiquidity()` is still incompatible with calls from the gauge, but this requirement is still not explicit.

7.2 Pool Can Be Initialized With Considerably Wrong Price

Informational | Version 1 | Acknowledged

CS-VELOSLIP-007

The unpermissioned `createPool()` function of `CLPoolFactory` allow anyone to create and initialize a pool. The initialization `sqrtPriceX96` of the pool determines where the `tick` of the pool is initialized, and consequently the starting point of the first swap. The pool creator can therefore sets the `tick` far from the actual price, up to `1774544` ticks away from the actual price (`TickMath.MAX_TICK * 2`), however in general the distance can be set at most `MAX_TICKS` away from the actual price (`887272`). If the pool tickSpacing is `1`, even with the `tickBitmap`, that would require `MAX_TICKS/256 == 3465` SLOADs to reach the correct tick, costing over 8M gas (without accounting for the gas expenditure of the other operations in the loop). This opens a griefing attack vector, where the attacker uses about 300K gas for pool creation, and the first swapper incurs a gas cost greater than 8M gas.

Acknowledged:

Velodrome has acknowledged this issue and replied with:

"Prices for new pools become stale quickly even for pools initialized at the correct price. If there is no/little liquidity, it becomes easy to move the price to a different tick, so it is considered an acceptable risk."

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 Attackers Can in the Future Generate Address Collisions

Note Version 1

Possible collisions of addresses are present through the codebase, some of the collisions can be found with a realistic computational budget today, and some others are still very expensive but will become feasible in the next decades, as computational power becomes cheaper.

- The salt used in create3 deployment by root and leaf gauge factories is a value of only 11 bytes (88 bits). The salt for gauge deployment is computed as the first 11 bytes of the keccak of pool parameters (`keccak(chainId ++ token0 ++ token1 ++ tickSpacing)`). We can find two pool configurations which produce the same salt with ~50% probability requires only sampling 2^44 pool configurations (randomizing over token0 for example), because of the Birthday paradox. Trying 2^44 attempts is feasible with modest computational power at the current day. This allows to create two pools whose gauges will have the same address, only one of the gauges can be therefore successfully created. Since on each pool one of the tokens is random, pools with couples of "real" tokens are not affected.
- Finding a collision between 160 bits addresses (generating the same address in two different ways, for example create2 and EOA) requires on the order of 2^80 tries. It is feasible with a computational budget in the order of billions of dollars (currently roughly estimated around $3B). The relative ease of address collision generation is the rationale for EIP-3607. More information on the cost of finding a collision can be found in the following Sherlock issue https://github.com/sherlock-audit/2024-06-makerdao-endgame-judging/issues/64 or in the document by Ethereum Foundation researcher Dmitry Khovratovich here . In the present Velodrome Superchain system, collisions can be generated for precomputed addresses, which include Leaf and Root pools, and Leaf and Root gauges. A possible exploit using an address collision for leaf gauges could be the following:
    - attackers find a way to create a malicious contract at address `X`, and creates a leaf pool `B` whose precomputed gauge is also address `X`. attacker does not deploy those yet.
    - attacker persuades Velodrome to create a gauge for pool `B`. The Root gauge for `B` is created, and a `CREATE_GAUGE` message is communicated to the bridge.
    - while the `CREATE_GAUGE` message has been processed by the root chain, but not the leaf chain, attacker creates malicious contract `X`.
    - `CREATE_GAUGE` is received on the leaf, and reverts because a contract already exists at `X`.

  The previous example would cause the root and leaf chains to be out of sync, and the following `DEPOSIT` messages for gauge `X` to fail on leaf but not on root, breaking deposits and withdraws through the bridge (since one reverting transaction on leaf blocks all the following ones).

Precomputed addresses which could be the target of collision attacks are the addresses of leaf and root pools (create2, 256 bits salt), and leaf and root gauges (create3, 88 bits salt). As mentioned, the computational budget for such an attack would today be ~$3B. However if Moore's law remains valid, over the next 30 years the cost will be reduced by an order of 1000x.

8.2 Weekly Gauge Cap Can Limit Rewards Accumulated Over Multiple Weeks

Note Version 1

The reward amount distributed by `notifyRewardAmount()` is capped by `RootCLGaugeFactory.calculateMaxEmissions()`, a function calculates the maximum possible emission for a gauge in a week. If the reward accumulates over multiple weeks and before `notifyRewardAmount()` is called on the Gauge, the accumulated amount gets capped with the maximum emission of only one week. However, in practice we can assume that reward distribution is performed every week, since both voters and stakers are incentivized to trigger it.