PUBLIC

Code Assessment

of the Superchain diff Smart Contracts

05 Nov, 2024

Produced for

velodrome

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 9
4 Terminology 10
5 Findings 11
6 Resolved Findings 12
7 Informational 14
8 Notes 16

1 Executive Summary

Dear Velodrome team,

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

Velodrome Superchain is an extension of existing Velodrome V2 Optimism reward mechanics to other networks within the Optimism Superchain ecosystem. Deployments on all leaf chains are managed from the Optimism mainnet Root chain, via Hyperlane bridge infrastructure.

The most critical subjects covered in our audit are functional correctness, access control and front-running prevention. Security regarding all the aforementioned subjects is high. However, one minor issue was found, that might endanger the system, if access control restrictions are ever softened, see CreateX entropy bit limit.

The general subjects covered are code complexity, upgradeability and unit testing. Code complexity is low due to the use of standard libraries and good readability. Security regarding Upgradeability is good. Testing is good, but the compiler version is not fixed for tests and deployment, see Floating pragma version.

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	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1
• Specification Changed	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 diff 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	45bfd414892adabd95103669345418d4080fb4bc	Initial Version
2	5 Nov 2024	a4e56fd6e76fc05799501b0937c5f01e382d53fb	Version with fixes

For the solidity smart contracts, the compiler version `0.8.27` was chosen.

This review assumes that the `src/mainnet` contracts will be deployed to Optimism mainnet chain. Other contracts are assumed to be deployed EVM equivalent chains with Hyperlane bridge support. Prior to deploying to any Leaf network, it is assumed that compatibility tests will be performed.

The following contracts in the folder `contracts/src` are in the scope of the review:

```
Router.sol

extensions:
    ModeRouter.sol

pools:
    Pool.sol
    PoolFactory.sol
    PoolFees.sol
    extensions/ModePool.sol
    extensions/ModePoolFactory.sol

gauges:
    LeafGauge.sol
    LeafGaugeFactory.sol

rewards:
    BribeVotingReward.sol
    FeesVotingReward.sol
    Reward.sol
    VotingReward.sol
    VotingRewardsFactory.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.

In this report, we assume the previous version of Velodrome is safe, and the review is focused on reviewing the Superchain update changes.

The previous version of the protocol were taken at commit hash:

```
src/pools: dbb3cbcd28bbb3b5e693e47a31496ffed8e73adb
src/gauges: bf1793476eea9464b378fcc3bc0190f58bff76fe
src/rewards: 29d79b2ff38c70b7a9125a5deadaaefcab3adb75
Router.sol: 5094f4f197592985f35375c4ab7f0eb5916834c7
```

Third party integrations and libraries such as *OpenZeppelin* and `Hyperlane` are assumed to be secure and are not reviewed in this report. Configurable parameters of the system are out of scope and assume to be chosen correctly.

In Version 2, the scope has been modified as follows:

- The file and contract `BribeVotingReward` has been renamed `IncentiveVotingReward`.
- The file and contracts `ModeLeafGauge`, `ModeLeafGaugeFactory`, `ModeFeeSharing` and `CustomFeeModule` have been added to the scope. These are Mode network version of contracts that facilitate Sequencer fee sharing.

2.1.2 Trust model

Some functionality in the system is controlled by parties with elevated privileges. For example:

1. `governor` of the `Voter` contract.
2. `owner` of `LeafMessageBridge` and `RootMessageBridge` contracts.
3. `owner` of `LeafHLMessageModule` and `RootHLMessageModule` contracts.
4. `owner` of `XERC20` contract.

All such parties are assumed to be trusted and to act in a non-malicious manner."

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.

2.2.1 Brief Velodrome V2 overview

Velodrome Superchain is an extension of the Velodrome V2 AMM protocol. In Velodrome V2, liquidity pools for asset pairs are created on Optimism mainnet blockchain. Liquidity providers can deposit assets into these pools and get LP tokens in return. These LP tokens can be used to redeem the deposited assets and a share of the trading fees generated by the AMM protocol. LP tokens can be staked in `Gauge` contract to earn part of weekly VELO emissions. VELO tokens can be locked in `VotingEscrow` contract in exchange for veVELO EIP-721 tokens. Each veVELO token has a voting power that can be used on weekly VELO emissions distribution between different pools.

2.2.2 Velodrome Superchain

Velodrome Superchain effectively makes the V2 system multi-chain. Optimism mainnet is considered as the Root chain, and other chains are considered as Leaf chains.

Leaf chain contracts:

- `Pool` - Liquidity pool for an asset pair.
- `PoolFactory` - Factory contract for creating pools.

- `LeafGauge` - Staking contract for LP tokens.
- `LeafGaugeFactory` - Factory contract for creating `LeafGauge` contracts.
- `FeesVotingRewards` - Contract that receives trading fees for staked LP tokens and distributes them to the veVELO voters.
- `BribeVotingRewards` - Contract that receives and distributes extraordinary incentive rewards to the veVELO voters.
- `VotingRewardsFactory` - Factory contract for creating `FeesVotingRewards` and `BribeVotingRewards` contracts.
- `LeafHLMessageModule` - Contract for receiving messages sent from Root via Hyperlane.
- `LeafVoter` - Contract for creating, killing and reviving gauges on the Leaf chain, whitelisting tokens, claiming rewards from the `LeafGauge` contracts

Root chain contracts:

- `RootPool` - Contract that mirrors the `Pool` contract on the Leaf chain. Does not support swaps or liquidity deposits. Does not have same address as the corresponding `Pool` contract on the Leaf chain.
- `RootPoolFactory` - Factory contract for creating `RootPool` contracts. Has same address as `PoolFactory` contract on the Leaf chain.
- `RootGauge` - Root chain side `LeafGauge` contract. Has same address as `LeafGauge` on the Leaf chain and forwards calls to the corresponding `LeafGauge` contract using the Hyperlane protocol.
- `RootGaugeFactory` - Factory contract for creating `RootGauge` contracts. Has same address as `LeafGaugeFactory` contract on the Leaf chain.
- `RootFeesVotingRewards` - Root chain side `FeesVotingRewards` contract. Forwards calls to the corresponding Leaf `FeesVotingRewards` contract.
- `RootBribeVotingRewards` - Root chain side `BribeVotingRewards` contract. Forwards calls to the corresponding Leaf `BribeVotingRewards` contract.
- `RootVotingRewardsFactory` - Factory contract for creating `RootFeesVotingRewards` and `RootBribeVotingRewards` contracts. Has same address as `VotingRewardsFactory` contract on the Leaf chain.
- `RootHLMessageModule` - Contract for sending messages between chains via Hyperlane protocol.
- `Voter` - Existing Velodrome V2 Voter contract.

`XERC20` token will be used to distribute staking rewards in the Superchain. It is an ERC20 token with additional functions to transfer tokens between chains. `Velo` reward tokens will be locked in the `XERC20Lockbox` contract and new `XERC20` will be minted. `XERC20` contracts will be deployed on both Root and Leaf chains to the same address via `XERC20Factory` contract. `RootHLMessageModule` can burn `XERC20` and `LeafHLMessageModule` can mint them on the Leaf chain. This happens when e.g. Root `Voter.distribute()` function sends `NOTIFY_WITHOUT_CLAIM` or `NOTIFY` messages from Root to Leaf. Velodrome Superchain `XERC20` implementation also has rate limiting functionality, that restricts the amount of minted tokens over time. In addition, `TokenBridge` contract, deployed on both chains, can be used to transfer `XERC20` tokens between chains, by burning them on one chain and minting on another.

2.2.3 Pool and Gauge deployment flow

On Root chain `RootPoolFactory` needs to deploy `RootPool` contract using `create2` with salt derived from the pool tokens and Leaf chain id. On Leaf chain, a `Pool` contract can be deployed using `PoolFactory` contract. To stake LP tokens of this pool, a `Voter.createGauge()` function needs to be called on the Root chain. This call results in (effects only):

1. Deploys `RootBribesVotingRewards` and `RootFeesVotingRewards` contracts on the Root chain by calling `RootVotingRewardsFactory.createRewards()`.
2. Deploys `RootGauge` contract on the Root chain by calling `RootGaugeFactory.createGauge()`. This creates `CREATE_GAUGE` message that is sent to the Leaf chain via Hyperlane protocol.

On a leaf chain, processing of the `CREATE_GAUGE` message results in:

1. Deployment of the `Pool` contract, if it is not already deployed.
2. Deployment of `FeesVotingRewards` and `BribeVotingRewards` contracts on the Leaf chain by calling `VotingRewardsFactory.createRewards()`.
3. Deployment of `LeafGauge` contract on the Leaf chain by calling `LeafGaugeFactory.createGauge()`.

As a result, the Gauge created on a Root chain is duplicated on a Leaf chain with the same address, but different functionality.

2.2.4 Inter-chain communication

All inter-chain communication is done via Hyperlane protocol. System contracts on root chain call `RootMessageBridge.sendMesssage()` function, that depending on the destination chain, dispatches the message to appropriate module such as `RootHLMessageModule`. `RootHLMessageModule` performs communication protocol specific tasks and calls. In our case - sends the message to `LeafHLMessageModule` via Hyperlane protocol.

`LeafMessageBridge` on the Leaf chain is a contract sole purpose of which is to store `module` that is allowed to handle messages sent to the leaf chain. This `module` is assumed to be `LeafHLMessageModule`.

2.2.5 Mode Chain contracts

Mode Chain has a sequencer fee sharing mechanism. Contract developer can register their contracts in the Sequencer Fee Sharing (SFS) contract to receive a share of the sequencer fees. The `ModePoolFactory` constructor performs the registration and mints an NFT token that can be used to claim the fees. Pool contracts deployed by the factory assign themselves to this NFT.

2.3 Changes in Version 2

- The custom swap fee is now set in an external module `CustomFeeModule` and retrieved via call from the `Factory`. The factory accepts a maximum custom fee of 10% (instead of 3%).
- New Gauge contracts for Mode Network only (`ModeGauge` and `ModeGaugeFactory`) support the sequencer fee sharing mechanism.
- The Reward contract on Leaf use the timestamp of Optimism Mainnet instead of the block timestamp of the Leaf Chain for reward distribution.

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

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	0
Low-Severity Findings	1

- Pool Not Fully Compatible to Be Used as Proxy Implementation Specification Changed

Informational Findings	1

- getAmountsOut Returns Zero on Invalid Swap Code Corrected

6.1 `Pool` Not Fully Compatible to Be Used as Proxy Implementation

Design Low Version 1 Specification Changed

CS-VELO-SC-DIFF-001

`Pool` contract inherits `EIP712` via `ERC20Permit`. `PoolFactory` contract deploys minimal proxy clone of `Pool` contract and relies on `Pool.initialize()` to set the storage variables. However, some fields of the `EIP712` contract are defined as immutable, and thus will be the same for all clones of the `Pool` contract. Following issues are present in the current implementation:

1. Getter `eip712Domain()` will return empty string field `_name`, due to how `_EIP712Name()` is implemented in the `Pool` contract.
2. `_buildDomainSeparator()` will use immutable `_hashedName` that is `keccak256(bytes(""))`. `DOMAIN_SEPARATOR()` getter will return result of `_buildDomainSeparator()`.
3. `_domainSeparatorV4()` has internal caching optimization that will never be triggered when minimal proxy is used.

Specification Changed:

In Version 2, the `Pool` no longer inherits `ERC20Permit` and `EIP712`. This solves the compatibility issue.

6.2 getAmountsOut Returns Zero on Invalid Swap

Informational Version 1 Code Corrected

CS-VELO-SC-DIFF-006

The function `Router.getAmountsOut` calculates the amount of tokens returned from multiple swaps. However, when a user provides a route without a pool deployed by the factory, the function will return zero tokens instead of reverting.

```
amounts[0] = amountIn;
uint256 _length = routes.length;
for (uint256 i = 0; i < _length; i++) {
    address pool = poolFor(routes[i].from, routes[i].to, routes[i].stable);
    if (IPoolFactory(factory).isPool(pool)) {
        amounts[i + 1] = IPool(pool).getAmountOut(amounts[i], routes[i].from);
    }
}
```

As a result, any other application calling `getAmountsOut` could continue with zero amount out when provided an invalid route. For example, function `swapExactTokensForETH` and `swapExactETHForTokens` use `getAmountsOut` to calculate the amount of tokens returned from multiple swaps. Provided an invalid route, the function will continue execution and only revert later when a call to function `swap` on a non-existent pool address will be attempted due to Solidity extcodesize check.

Code corrected:

The function `Router.getAmountsOut` has been updated to revert when a pool provided was not deployed by the factory.

```
if (IPoolFactory(factory).isPool(pool)) {
    amounts[i + 1] = IPool(pool).getAmountOut(amounts[i], routes[i].from);
} else {
    revert InvalidPath();
}
```

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 CreateX Entropy Bit Limit

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

CS-VELO-SC-DIFF-002

The Gauge factories use CreateX, relying only on 11 bytes (88 bits) of entropy, for creating a gauge for a (token0, token1) pair.

```
bytes32 salt = keccak256(abi.encodePacked(_chainid, _token0, _token1, _stable));
bytes11 entropy = bytes11(salt);

gauge = CreateXLibrary.CREATEX.deployCreate3({
    salt: entropy.calculateSalt({_deployer: address(this)}),
```

An attacker could theoretically deploy a new token with a deterministic address (using create2) and then create gauge with the same salt as another pool pair. This stops any gauge being created for this pool pair as the address is "taken" by the other pool.

Colliding any (token0, token1) to any(token0, token1) has around 2^44 bits of entropy for 40% probability of collision. However, since these are random tokens, the consequences are minor.

The worst case scenario for collision is to create some collision with a pool like WETH-USDC, preventing gauge creation. This would require to brute-forcing 88 bits, which is not feasible with vast computational resources.

Further, gauge creation is currently permissioned, making this attack vector practically impossible

---

Acknowledged:

Velodrome is aware of this behavior, but has decided to keep the code unchanged.

7.2 Fees and Bribe Addresses on Leaf and Root Chain

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

CS-VELO-SC-DIFF-003

There is a strong invariant that the contract addresses of the "same contracts" are the same on Root and Leaf chains. However, Fees and Bribes contracts can have the same addresses but belong to completely different Gauges. For example, all deployed `FeesVotingReward` with nonce N on all the EVM equivalent chain (both Root and Leaf) will have same addresses. Users of the protocol could make incorrect assumptions about contract addresses being equal.

Acknowledged:

Velodrome answered:

```
Risk is understood. Additional documentation will be added to inform users that there are no invariants around fees and bribe addresses on leaf or root.
```

7.3 Floating Pragma Version

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

CS-VELO-SC-DIFF-004

The Superchain contracts use a floating pragma *>=0.8.19 <0.9.0*. Several versions in that range have known vulnerabilities: https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json Project has a fixed solidity version *0.8.27* compiler version in foundry with no known vulnerabilities. However, hardhat `hardhat.config.js` sets the solidity version to *0.8.24*. This discrepancy can lead to potential issues in the future, due to one version being used for testing and another for deployment. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using a compiler version that might introduce bugs that affect the contract system negatively.

Acknowledged:

Velodrome has acknowledged the issue and noted that:

```
Code versions will be synchronized across testing environments.
```

7.4 Misleading Names

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

CS-VELO-SC-DIFF-005

Certain variable and contract names may be misleading, causing confusion for developers and users. This issue is informational and does not represent a security vulnerability.

Instances of this issue are:

1. The contract `VotingReward` stores the bridge address in state variable `authorized` to protect access to certain functions (i.e. deposit or withdraw). However, the variable name authorized can be misleading, as this term is used in other contracts for the Voter and other admin function and does not imply that it can be called by a bridge contract.
2. Most of the contracts on the Root side have a `Root` prefix. E.g. `RootGauge`, `RootVotingRewardsFactory`. However, not all Leaf side contracts have a `Leaf` prefix. For example, `VotingRewardsFactory` and `Pool`. Making the naming more explicit can improve readability and reduce confusion.

Acknowledged:

Velodrome has acknowledged the issue, but has decided to keep the code unchanged.

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 Reproducible Deployment

Note Version 1

Superchain contracts need to be deployed to many different chains and environments. To ensure verifiable and reproducible deployments, consider the following recommendations:

- **Compiler Version**: Fix the compiler version to avoid inconsistencies. Avoid using floating pragmas (e.g., *pragma solidity ^0.8.0;*) or explicitly set the version in the configuration.
- **Compiler Options**: Define and commit key options such as optimizations and the EVM version. The deployment strategy (e.g., gas vs. execution optimization) should be consistent across environments.
- **Library Versions**: If you are using external libraries (e.g., via NPM), ensure that specific versions are fixed. Avoid floating versions that can change over time. Currently used "@hyperlane-xyz/core" is fixed.
- **Build System Version**: Ensure that the version of your build system (e.g., Hardhat, Foundry, Truffle, etc.) is specified to avoid inconsistencies due to changes in defaults or features between versions. Optionally, export or document the complete configuration to ensure all details are explicit and reproducible.
- **Deployment Files/Logs**: Keep logs and build files of the deployment process, including transaction hashes and addresses, to support traceability.
- **Tagging**: Tag the commit used for the deployment to create an easy reference for future audits and verification.

Following these guidelines will help to prove that the deployed bytecode matches the source code using e.g. Etherscan or Deployment Validation.