PUBLIC

Code Assessment

of the Superchain Interoperability Smart Contracts

20 May 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 Superchain Interoperability according to Scope to support you in forming an opinion on their security risks.

Velodrome implements an expansion of Velodrome AMM system to Superchain. With this expansion, the VELO rewards and incentives become available on chains beyond Optimism with the help of Hyperlane.

The most critical subjects covered in our audit are multi-chain state consistency, functional correctness, Hyperlane integration, and frontrunning resistance.

The mechanisms ensuring consistent state across chains are robust and deliver a high level of security. The functional correctness is high as issues such as Voting period in epochs can be bypassed by using poke() have been resolved. Similarly, the integration with Hyperlane's bridging mechanism was found to be correct after issues such as Metadata Misuse in Bridges were resolved.

In the second version of the codebase, the mechanism enforcing the ordering of specific types of messages (DEPOSIT and WITHDRAW) was relaxed. This could lead the state of some contracts to be temporarily inconsistent which would lead to accounting issues (Voting power can be temporarily artificially inflated). The issue has been addressed, but it should be noted that the system relies heavily on the assumption that messages from the root to the leaf will be processed within 1 hour.

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	2
• Code Corrected	2
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	8
• Code Corrected	4
• Risk Accepted	3
• Acknowledged	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 Interoperability 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 October 2024	45bfd414892adabd95103669345418d4080fb4bc	Initial Version
2	5 November 2024	2daa9e303cad0c907da7dbfadf4d0ff1bbd18aa0	Fixes
3	6 November 2024	fa572a3c5b0828333cbcf74797848b9d90300e47	First report
4	5 February 2025	5d3f5a87596043b9a6eb4d34399a012e1cfeb9d3	Gas router, domains
5	20 February 2025	bff89f808e3d33bb3782e5228d6b94aaaf3a43ff	Second report
6	20 April 2025	79a3145441b94b4c38de5c274026d5396db680ab	Superwap Router
7	16 May 2025	c15a81e1f9faffb721563826b359b05ee78afb84	Fixes

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

This review assumes that the `src/root` contracts will be deployed to Optimism mainnet chain. Other contracts are assumed to be deployed on 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 `src/` are in the scope of the review:

```
bridge:
    extensions:
        hyperlane:
            ModeLeafHLMessageModule.sol

        ModeLeafMessageBridge.sol
        ModeTokenBridge.sol

    hyperlane:
        LeafHLMessageModule.sol

    ChainRegistry.sol
    CrossChainRegistry.sol
    LeafMessageBridge.sol
    TokenBridge.sol

extensions:
    ModeFeeSharing.sol

gauges:
    extensions:
        ModeLeafGauge.sol
```

```
        ModeLeafGaugeFactory.sol

    LeafGauge.sol
    LeafGaugeFactory.sol

libraries:
    rateLimits:
        RateLimitedMidpointLibrary.sol
        RateLimitMidpointCommonLibrary.sol

    Commands.sol
    CreateXLibrary.sol
    GasLimits.sol
    VelodromeTimeLibrary.sol

root:
    bridge:
        hyperlane:
            RootHLMessageModule.sol
        RootMessageBridge.sol

    emergencyCouncil:
        EmergencyCouncil.sol

    gauges:
        RootGauge.sol
        RootGaugeFactory.sol

    pools:
        RootPool.sol
        RootPoolFactory.sol

    rewards:
        RootIncentiveVotingReward.sol
        RootFeesVotingReward.sol
        RootVotingRewardsFactory.sol

voter:
    extensions:
        ModeLeafVoter.sol
    LeafVoter.sol

xerc20:
    MintLimits.sol
    XERC20.sol
    XERC20Factory.sol
    XERC20Lockbox.sol
```

Note that the scope represents the files in Version 3. The project layout has changed since Version 1.

In Version 4, the scope was updated as follows:

Added contracts:

```
bridge:
    BaseTokenBridge.sol
```

```
    LeafEscrowTokenBridge.sol
    LeafTokenBridge.sol
    LeafRestrictedTokenBridge.sol

root:
    bridge:
        hyperlane:
            GasRouter.sol
            Paymaster.sol
            PaymasterVault.sol
        RootEscrowTokenBridge.sol
        RootTokenBridge.sol
        RootRestrictedTokenBridge.sol

xerc20:
    extensions:
        RestrictedXERC20.sol
        RestrictedXERC20Factory.sol
```

Removed contracts:

```
bridge:
    TokenBridge.sol
```

Renamed contracts:

```
bridge:
    extensions:
        ModeTokenBridge.sol into ModeLeafEscrowTokenBridge.sol
```

In Version 6, only the changes made to the following scope was reviewed:

```
bridge:
    BaseTokenBridge.sol
    DomainRegistry.sol
    LeafEscrowTokenBridge.sol
    LeafTokenBridge.sol

root:
    bridge:
        RootEscrowTokenBridge.sol
        RootTokenBridge.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 and integrations, such as OpenZeppelin and Hyperlane, are explicitly out of the scope of this review. They are assumed to be secure and conform to their specification. Hyperlane can make use of user-defined hooks. These are assumed to be implemented correctly and not expose the system in scope to additional threats. In this report, we assume the previous version of Velodrome is safe, and the review is focused on the interoperability of the superchain contracts.

The previous version of the protocol was taken at commit hash: `ee15bd1e63d3b33ce8d179f73bca7390812bd99b` which is v2.1.

Configurable parameters of the system such, as gas limits, are out of scope and assumed to be chosen correctly by the system admins. Admin actions are assumed to not expose the system to potential security threats.

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 Superchain is an expansion of existing Velodrome V2 AMM protocol capabilities.

In Velodrome V2, liquidity pools for asset pairs are created on Optimism mainnet. 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. Each week is called an epoch and the amount of `VELO` emissions is distributed based on the number of votes a gauge receives. `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 to steer which gauges corresponding to certain pools receive `VELO` distributions for the following week. In return for controlling the `VELO` distributes, these voters earn the fees for those pools from the prior epoch, as well as any incentives that were deposited by users.

Velodrome Superchain extends the capabilities of Velodrome V2 to support reward distribution, voting and bribes across Optimism Superchain network.

2.2.1 Velodrome V2 Core Components

Velodrome V2 protocol, deployed on Optimism, comprises the following core components:

- Pools: These could be constant product AMMs similar to UniswapV2, with a different fee for stable and volatile pairs, or concentrated liquidity pools similar to UniswapV3. For constant product AMMs, LPs deposit the two liquidity tokens and mint some LP tokens. For the concentrated liquidity ones, they mint an NFT.
- PoolFactory: It is a factory contract that deploys pools. It is also responsible for storing the fee that pools should charge for swapping.
- Gauges: LP tokens can be staked here to earn `VELO` emission rewards while relinquishing the earned LP fees.
- VotingEscrow: `VELO` holders can lock tokens and earn a `veVELO` NFT in return. The lock period defines how much voting power the `veVELO` token will have.

  Conversion formula is: $veVELO\ voting\ power = locked\ VELO\ amount \cdot \frac{max(lock\ period, 4\ years)}{4\ years}$.
- Voter: the contract that handles votes, emission distribution and creation of gauges, voting rewards and bribes contracts.
- BribeVotingReward: It stores the bribes for a specific gauge. Anyone can deposit an amount on any whitelisted token as a bribe. Users who voted for this particular gauge in `Voter` can claim these rewards.
- FeeVotingReward: It stores the claimed trading fees of a pool. `veVELO` voters can claim these rewards.

2.2.2 Superchain Protocol Components

Velodrome Superchain effectively makes the V2 system multi-chain, by bridging the VELO distribution, voting and bribes across Optimism Superchain network. Optimism mainnet is considered as the Root chain, and other chains are considered as Leaf chains.

Each Leaf chain will have its own set of core components, similar to Velodrome V2 core components. However, the Leaf pools and gauges will be replicated on Root chain. This way Root chain will have a complete view of the state of all the pools and gauges across all chains and act as a synchronization point.

Leaf specific components are:

- `Pool` - Liquidity pool for an asset pair.
- `PoolFactory` - Factory contract for creating pools.
- `LeafGauge` - Staking contract for LP tokens. It receives `xVELO` rewards from the Root chain and LP stakers can claim them.
- `LeafGaugeFactory` - Factory contract for creating `LeafGauge` contracts.
- `FeesVotingReward` - Contract that receives trading fees for staked LP tokens and distributes them to the `veVELO` voters.
- `BribeVotingReward` - Contract that receives and distributes extraordinary incentive rewards to the `veVELO` voters.
- `VotingRewardsFactory` - Factory contract for creating `FeesVotingReward` and `BribeVotingReward` contracts.
- `LeafHLMessageModule` - Contract for receiving messages sent from Root via Hyperlane.
- `LeafMessageBridge` - Contract that keeps track of the module that handles messages from the Root chain. Assumed to be `LeafHLMessageModule` for this assessment.
- `LeafVoter` - Contract for creating, killing and reviving gauges on the Leaf chain, whitelisting tokens, claiming rewards from the `LeafGauge` contracts
- `XERC20` - `ERC20` token that represents `VELO` on the Leaf chains. It is an `ERC20` token that supports token transfers between chains.

Root chain contracts:

- `RootPool` - Contract that mirrors the `Pool` contract on the Leaf chain. Does not support swaps or liquidity deposits.
- `RootPoolFactory` - Factory contract for creating `RootPool` contracts. It has the same address as the `PoolFactory` contract on the Leaf chain.
- `RootGauge` - Root chain size `LeafGauge` contract. It has same address as `LeafGauge` on the Leaf chain and forwards reward notification messages to the corresponding `LeafGauge` contract using the Hyperlane protocol. LP stakers cannot claim `VELO` rewards from this contract.
- `RootGaugeFactory` - Factory contract for creating `RootGauge` contracts. It has the same address as the `LeafGaugeFactory` contract on the Leaf chain.
- `RootFeesVotingReward` - Root chain size `FeesVotingReward` contract. Forwards calls to the corresponding Leaf `FeesVotingReward` contract.
- `RootBribeVotingReward` - Root chain size `BribeVotingReward` contract. Forwards calls to the corresponding Leaf `BribeVotingReward` contract.
- `RootVotingRewardsFactory` - Factory contract for creating `RootFeesVotingReward` and `RootBribeVotingReward` contracts. It has the same address as the `VotingRewardsFactory` contract on the Leaf chain.

- `RootHLMessageModule` - Contract for sending messages to Leaf chain via Hyperlane. Forwards messages to the corresponding `LeafHLMessageModule` contract.
- `RootMessageBridge` - Contract that keeps track of the module that handles outgoing messages. Assumed to be `RootHLMessageModule` for this assessment.
- `Voter` - Existing Velodrome V2 Voter contract.
- `XERC20` - cross-chain `VELO` variant, minted by `XERC20Lockbox`
- `XERC20Lockbox` - Contract that holds wrapped `VELO` tokens on the Root chain
- `EmergencyCouncil` - Ownable contract that can kill or revive both Velodrome V2 and Superchain Gauges.

RootMessageBridge and LeafMessageBridge with Message Modules

`RootMessageBridge` with `RootHLMessageModule` is deployed on the Root chain and handles dispatching messages to the Leaf chain. `LeafMessageBridge` with `LeafHLMessageModule` are deployed on the Leaf chain and handle incoming messages from the Root chain. Communication is one way, from the Root to the Leaf chain. `RootMessageBridge` supports nine message commands, each designed to manage operations on the Leaf chain effectively:

- `DEPOSIT` - Triggered when `Voter.vote()` is called on the Root chain. It increases the user's voting balance in `FeesVotingRewards` and `BribeVotingRewards` of the target `LeafGauge`.
- `WITHDRAW` - Triggered when `Voter.reset()` is called on the Root chain or as a cleanup procedure at the start of `Voter.votes()`. It lowers the user's voting balance in `FeesVotingRewards` and `BribeVotingRewards` of the target `LeafGauge`.
- `NOTIFY` - Triggered when `Voter.distribute()` is called on the Root chain. It notifies the gauge about the rewards available for claiming.
- `NOTIFY_WITHOUT_CLAIM` - Notify the gauge about the rewards without claiming Pool trading fees.
- `GET_INCENTIVES` - Claim bribes from Leaf chain `BribeVotingRewards` contract.
- `GET_FEES` - Claim fees from the Leaf `FeeVotingRewards` contract.
- `CREATE_GAUGE` - Create a new `LeafGauge` on the Leaf chain.
- `KILL_GAUGE` - Kills a `LeafGauge`. Can only be initiated by `EmergencyCouncil`.
- `REVIVE_GAUGE` - Revives a `LeafGauge`. Can only be initiated by `EmergencyCouncil`.

Each `DEPOSIT` and `WITHDRAW` command for a specific Leaf chain includes a nonce. This nonce makes incoming `DEPOSIT` and `WITHDRAW` messages on the Leaf chain to be processed in order. This prevents situations, where the sum of user votes on Leaf chains temporarily exceeds the Root chain voting power. Note that the use of nonces was abandoned in the next iterations of the codebase.

XERC20

An `XERC20` token is an ownable, mintable/burnable `ERC20` token. Used to wrap `VELO` into `xVELO`. On Root chain `VELO` tokens are locked in `XERC20Lockbox` and `XERC20` tokens are minted. An owner can assign a cap to any `bridge` address, that would enable it to mint/burn `XERC20` tokens. The $cap/2$ value is the midpoint of the buffer. Buffer value decreases or increases linearly over time until the midpoint is reached. When `XERC20` tokens are minted buffer is decreased by the amount minted. When `XERC20` tokens are burned buffer is increased by the amount burned. Transaction is reverted if the buffer is below zero or above the cap.

In addition to the standard `XERC20` interface, mint and burn that were already discussed, the contract exposes the following interface:

- `setBufferCap()`: the owner of the contract sets a new non-zero buffer cap for an address with a buffer cap already specified. The cap should be above `1_000e18`.

- `setRateLimitPerSecond()`: the owner of the contract sets a new rate limit for an already registered address. The rate limit cannot exceed a `25_000e18`.
- `addBridge()`: the owner instantiates the buffer cap, and the rate limit for an address. The available buffer is initialized at the midpoint.
- `removeBridge()`: the owner deletes all the data related to a buffer of a specific address rendering the address unable to mint or burn.

Velodrome Superchain relies on Hyperlane protocol to transfer messages from Root to Leaf chains. A separate pair of Bridge contract, one on Root and one on Leaf, need to be deployed to facilitate cross-chain communication. For this assessment, two bridges are in scope. `TokenMessageBridge` - `RootMessageBridge`. However, `XERC20` can also be transferred through the optimism superchain bridge. Tokens can be minted and burned by the bridge via the `crosschainMint()` and `crosschainBurn` respectively.

TokenBridge

`TokenBridge` - Contract that facilitate `XERC20` token transfers between chains.

- Sending tokens: `TokenBridge.sendToken()` burns the specified amount of `XERC20` tokens on the origin chain, calls Hyperlane's `Mailbox.dispatch()`, and ensures the transaction is routed to the registered destination chain.
- Receiving Tokens: `TokenBridge.handle()` confirms that the message originates Hyperlane's `Mailbox` and mints the specified amount of `XERC20` tokens on the destination chain for the designated recipient.
- `ChainRegistry` functionality allows the owner to register and deregister chains that can be used as destinations for cross-chain token transfers.

Note that both minting and burning are constrained by the `XERC20` buffer cap.

2.2.3 Important Call Paths

Depositing on the Voting Rewards Contracts

1. `veVELO` holder calls `Voter.vote()` on the Root chain with list of pool and corresponding vote weights.
2. `Voter.vote()` calls `_reset()`. This "undoes" the previous vote by calling `_withdraw()` on the `RootFeesVotingReward` and `RootBribeVotingReward` contracts.
    - `RootBribeVotingReward._withdraw()` is a no-op.
    - `RootFeesVotingReward` sends a `WITHDRAW` message to the `LeafHLMessageModule` on the Leaf chain.
    - `LeafHLMessageModule` calls `FeeVotingReward._withdraw()` and `BribeVotingReward._withdraw()` on the Leaf chain.
3. `Voter.vote()` calls `_deposit()` on the Fee and Bribe reward contracts for the new pools.
    - `RootFeesVotingReward` sends a `DEPOSIT` message to the `FeesVotingReward` on the Leaf chain.
    - `RootFeesVotingReward` sends a `DEPOSIT` message to the `BribeVotingReward` on the Leaf chain.

Withdrawing from the Voting Rewards Contracts

1. `veVELO` holder calls `Voter.reset()` on the Root chain.
2. `Voter.reset()` calls `_reset()`. As described in the depositing flow above.

Getting Incentives/Bribes

1. veVELO holder calls `Voter.claimBribes()` on the Root chain, specifying the list of reward tokens to claim.
2. `Voter.claimBribes()` calls `RootBribeVotingReward.getReward()`. `RootBribeVotingReward` determines the recipient address of the rewards on the Leaf chain.

   It is either the owner of veVELO NFT or recipient registered in `RootVotingRewardsFactory`
3. `RootBribeVotingReward.getReward()` sends a `GET_INCENTIVES` message to the `LeafHLMessageModule` on the Leaf chain via the `RootMessageBridge`.
4. `LeafHLMessageModule` calls `BribeVotingReward.getRewards()` on the Leaf chain.
5. `BribeVotingReward.getRewards()` transfers the reward tokens to the recipient.

Getting Fees

1. A veVELO holder calls `Voter.claimFees()` on the Root chain, specifying the list of fee reward tokens to claim.
2. `Voter.claimFees()` invokes `RootFeesVotingReward.getReward()`. The `RootFeesVotingReward` contract determines the recipient address for the rewards on the Leaf chain, which is either the owner of the veVELO NFT or a recipient registered in `RootVotingRewardsFactory`.
3. `RootFeesVotingReward.getReward()` sends a `GET_FEES` message to the `LeafHLMessageModule` on the Leaf chain via the `RootMessageBridge`.
4. Upon receiving the message, `LeafHLMessageModule` calls `FeesVotingReward.getRewards()` on the Leaf chain.
5. `FeesVotingReward.getRewards()` then transfers the fee reward tokens to the designated recipient.

Pool and Gauge deployment flow

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 with different functionality.

Notifying Rewards

1. Any user can initiate distribution of claimable rewards once per week by calling `Voter.distribute()` on the Root chain.

2. `Voter.distribute()` calls `Minter.updatePeriod()` to mint new `VELO` tokens. The `RootGauge` is then approved to withdraw the minted `VELO` amount from `Voter`. This action is expected to occur once per epoch.
3. `VELO` tokens are locked on `XERC20Lockbox` and `xVELO` tokens are minted.
4. The `RootMessageBridge` burns the `xVELO` on Root chain and dispatches a `NOTIFY` message to the `LeafHLMessageModule` on the Leaf chain.
5. Upon receiving the `NOTIFY` message, the `LeafHLMessageModule` mints `xVELO` on the Leaf chain.

   `LeafGauge.notifyRewardAmount()` is called.
6. `LeafGauge.notifyRewardAmount()` triggers the gauge to claim trading fees from the pool and increases the reward rate of current epoch.

There exists an alternative process, where trading fees are not claimed from the pool. This starts when a notify admin calls `RootGauge.notifyRewardWithoutClaim()`. In this case `NOTIFY_WITHOUT_CLAIM` is bridged and `LeafGauge.notifyRewardWithoutClaim()` is called.

2.2.4 Deployment

`RootBribeVotingReward` and its Leaf chain equivalent `BribeVotingReward` are deployed by respectively the `RootVotingRewardFactory` and the `VotingRewardsFactory` with new keyword to create the contracts. The same factories also deploy the `RootFeesVotingReward` and `FeesVotingReward` contracts in the same way. Therefore, the reward contracts will be deployed on different addresses on the Root and the Leaf chains.

`RootPoolFactory` and `PoolFactory` will respectively deploy `RootPool` and `Pool` on the Root and Leaf chains. While the implementation of such pools will be deployed through CREATE3 as described above, the pool factories will deploy a new pool through `Clones.cloneDeterministic()`. This will ensure that the same pool is deployed on all chains. However, the pools will have different addresses on Root and Leaf as the `RootPoolFactory` uses the `chainid` as part of the salt whereas `PoolFactory` only uses `token0` and `token1` and `stable` as part of the salt.

`RootGaugeFactory` and `LeafGaugeFactory` both use `CREATEX.deployCreate3()` to deploy respectively `RootGauge` and `LeafGauge`. Both factories use the same salt parameters therefore the Root gauge will be at the same address as the Leaf gauge on the Leaf chain.

`LeafVoter` will also be deployed with CREATE3, but it is important to note that it will not have the same address as the `Voter` currently deployed on Optimism.

`XERC20` and `XERC20Lockbox` will be deployed on the Root chain while `XERC20` without a lockbox will be deployed on Leaf chains.

2.2.5 VERSION 2

The following changes were introduced in the second version of the codebase:

- `DEPOSIT` and `WITHDRAW` messages can only be emitted up until an hour before the end of the epoch unless the `tx.origin` is the owner or an approved party to use the token. A consequence of that is that after the end of the vote, users can only `poke()` themselves.
- Users cannot claim rewards in an epoch they already voted in.
- To mitigate any potential DoS issues, the use of nonces has been abandoned for `DEPOSIT` and `WITHDRAW` messages. Instead, the message includes the `block.timestamp` of the root chain. This change relaxes the invariant where deposits and withdraw messages are received in the same order they are emitted. In the current implementation, these messages can be reordered under the following assumptions:

- When voting, bridging of the deposit and the withdraw for a `tokenId` will eventually be processed before the epoch finishes. Therefore, even in the case of reordering the state will eventually be consistent. It is important to note however, that the state can be inconsistent during the processing of the messages e.g., two deposits might have occurred without a withdrawal.
- Rewards are only requested for past epochs and requesting the leaf rewards can only happen after the flip of an epoch where a specific `tokenId` has voted. This guarantees that both the withdraw and deposit messages will have already been processed.

- For `RootHLMessageModule` and `TokenBridge`, a custom hook is used when quoting or dispatching a message. In the case of the `TokenBridge`, users get refunded for the excessive gas. It is currently only meant to be used for users (EOAs) to transfer tokens between chains. This is why a `transfer` primitive is used to refund users. It will likely not be the only way, nor the final way for users to transfer tokens between chains.
- Some leaf contracts of the system have been extended to be able to receive rewards when deployed on the Mode rollup.

2.2.6 VERSION 3

The following changes were introduced in the third version of the codebase:

- `GET_FEES`/ `GET_INCENTIVES` is only allowed an hour after the start of a new epoch until one hour before the epoch end. This is done to ensure that all deposit and withdrawal messages of the previous epoch have been processed and that the state is consistent. This change heavily relies on the following assumption: All messages should be processed within one hour of their emission. If this assumption breaks the system might end up in an inconsistent state which some users can benefit from.
- `DEPOSIT` and `WITHDRAW` messages can now be sent to the leaf chain until the end of the epoch by either any user using `poke()` or by whitelisted users using `vote()`.

2.2.7 VERSION 4

The following changes were introduced in the fourth version of the codebase:

- A `GasRouter` is introduced to update the gas price of the commands instead of having them hardcoded. It maintains a mapping between commands and gas limits. The `GasRouter` is only used on the Root chain and its mapping is assumed to properly be populated.
- A mapping `chainID <-> hyperlane domain` has been added to support Hyperlane domains where `chainID != domain`. This mapping is used only on the root chain, as it is assumed that the leaf chains will only bridge to the root chain, where `chainID == domain`.
- When bridging `xVELO` from a leaf to the root chain, it is automatically redeemed into `VELO` and sent to the recipient on the root chain.
- It is now possible to bridge and lock `xVELO` from a leaf chain in one transaction. The `xVELO` is automatically redeemed for `VELO` and deposited in the `VotingEscrow` for a given `tokenId`. If the deposit fails, the `VELO` token is simply transferred to the provided recipient address.
- `xOP` Leaf Incentives: Velodrome wants to enable `OP` token to be wrapped in `xOP` and used with restrictions on a leaf chain. Users should be able to claim `xOP` on a leaf chain, but its utility is primarily intended to bridge it back to the root chain. Transfers on leaf chains are restricted to limit the token's broader usage. The same lockbox mechanism is employed to wrap `OP` to `xOP` as with `xVELO` (see Superchain interoperability report). `xOP` is bridged to a leaf chain using a `[Root]RestrictedTokenBridge`. If the destination chain is a chain other than BASE, the recipient should be a live gauge deployed on that chain. For BASE the recipient can be any

address including a gauge. Note that for the BASE case it is not possible to enforce further restrictions as the gauges on BASE are not known on the Root chain. In any case, if the recipient address is a gauge, `xOP` is directed to the contract that escrows the bribes/incentives after being minted for the bridge.

Transferring `xOP` is restricted on all leaf chains. Transfers can only originate from whitelisted addresses. An address is whitelisted only if it has received funds from the token bridge. Minting and burning the token is only restricted by the capped buffer mechanism already discussed earlier. As the bridge can directly burn `xOP`, all users are able to bridge back `xOP` to the root chain as long as they're able to interact with the bridge. It is important to note that any entity can mint or burn `xOP` should they be given a buffer. Such an entity could simulate transfers by burning for one address and minting for another one.

As with `xVELO`, `xOP` is unwrapped upon receiving from the `RootRestrictedTokenBridge`.

- Gas sponsoring: Velodrome enables gas sponsoring for the following two cases: 1) sending the `NOTIFY` command from the root to the leaf chains during the first hour of each new epoch and 2) for all the messages for a set of whitelisted addresses without further restrictions. Users (`tx.origin`) are whitelisted in `RootHLMessageModule`. Moreover, gas sponsoring is also available for a set of whitelisted `msg.sender` addresses on the `RootTokenBridge` when bridging `xVELO` from the root chain to a leaf chain. The whitelist is maintained in the `Paymaster` contract which is inherited by the bridge contracts. The funds used for sponsoring are escrowed in the `PaymasterVault`. The owner of the vault can withdraw assets, while the vault manager i.e., the Hyperlane module or the token bridge, can use the funds for sponsoring.

2.2.8 VERSION 6

In version 6, several updates were introduced to support the Velodrome Superswaps router's use of the bridge for asset bridging on behalf of users:

- The Hyperlane version was upgraded to 5.12.0.
- The `sendToken()` function was overloaded to accept a refund address. This allows the Velodrome Superswaps router to specify a refund address when sending tokens through the bridge, as the router cannot receive ETH transfers.
- Fee refund calculations were removed, as Hyperlane handles refunds internally and credits the specified refund address.
- To simplify chainID and Hyperlane domain conversions, the bridge now accepts a domain instead of a chainID. This change shifts the responsibility to the user to provide the correct domain for the destination chain.

2.2.9 VERSION 7

In version 7, only fixes for the issues found during the audit were introduced. No major changes were made to the system.

2.3 Trust Model

General:

The system is assumed to be deployed on the Optimism mainnet and the Superchain network.

VELO:

`VELO` emissions are distributed weekly and `Voter.distribute()` is called once per week.

Create3:

The following contracts are assumed to be deployed on the same address across all chains with CREATE3 where the address of the deployer is used in the salt:

- `RootPoolFactory` and `PoolFactory`
- `RootPool` and `Pool` implementation contracts. Proxies that use these implementations won't have the same address.
- `LeafGaugeFactory` and `RootGaugeFactory`
- `LeafGauge` and `RootGauge`
- `LeafHLMessageModule` and `RootHLMessageModule`
- `LeafMessageBridge` and `RootMessageBridge`
- `LeafEscrowMessageBridge` and `RootEscrowMessageBridge`
- `RootVotingRewardFactory` and `VotingRewardsFactory`
- `XERC20Factory`
- `XERC20`
- `TokenBridge`
- `LeafVoter`

Roles:

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

- `governor` of the `Voter` contract.
- `owner` of the `EmergencyCouncil` contract.
- `owner` of `LeafMessageBridge` and `RootMessageBridge` contracts.
- `owner` of `LeafEscrowMessageBridge` and `RootEscrowMessageBridge` contracts.
- `owner` of `LeafHLMessageModule` contract.
- `owner` of `XERC20` contract.
- `owner` of `ChainRegistry`, `CrossChainRegistry` and `TokenBridge`.

All such parties are assumed to be trusted and act in a non-malicious way or configure the system in a way that exposes it to threats.

Version 4:

For Version 4 we extend the trust model as follows:

- We assume that the infrastructure for `xOP` is going to be properly deployed and parametrized.
- For gas sponsoring:
  - The whitelist manager is assumed to properly configure the whitelist and the paymaster vault.
  - The whitelisted users are assumed to not try to drain the `PaymasterVault` by executing redundant transactions.

Tokens:

The system is expected to interact only with ERC-compliant tokens that are not malicious and present no specific risks or unusual behaviors (e.g., rebasing, transfers a different amount than requested, fee-on-transfer, double entry points, non-compliant interface, or hooks)

Hyperlane:

Hyperlane is out of scope for this assessment, however, the following assumptions are made:

- Hyperlane messages are assumed to be processed in less than 1 hour.

- The hook and ISM contracts used by the different bridges are assumed to properly ensure the security of the system, to validate the messages are genuine and cannot be replayed. If the configured hooks or ISM contracts are compromised, the bridge can be exploited to drain funds, or mint unbacked tokens.
- The non-mandatory hook used by the different token bridges is assumed to refund to the provided address the part of `msg.value` that is not used for fees.
- The different `GAS_LIMIT[_LOCK]()` constant functions value are assumed to be properly set to match as close as possible, but not lower than the actual gas cost of the transactions needed to relay the messages.

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 Open 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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	4

5.1 Reentrancy Due to `InterchainGasPaymaster` Calling `tx.origin`

Design Low Version 6 Risk Accepted

CS-VELO-SC-INOP-020

This issue extends the informational issue Potential reentrancy due to InterchainGasPaymaster calling tx.origin reported in Version 1 of the system with the new risk brought by the EVM Pectra hardfork.

The Hyperlane `Mailbox` might have `InterchainGasPaymaster` or other similar hooks as one of its default hooks. The function `payForGas()` of `InterchainGasPaymaster` is triggered as a post-dispatch hook.

```
function payForGas(
    bytes32 _messageId,
    uint32 _destinationDomain,
    uint256 _gasLimit,
    address _refundAddress
) public payable override {
    uint256 _requiredPayment = quoteGasPayment(
        _destinationDomain,
        _gasLimit
    );
    require(
        msg.value >= _requiredPayment,
        "IGP: insufficient interchain gas payment"
```

```
        );
        uint256 _overpayment = msg.value - _requiredPayment;
        if (_overpayment > 0) {
            require(_refundAddress != address(0), "no refund address");
            payable(_refundAddress).sendValue(_overpayment);
        }

        ...
    }
```

Since `RootHLMessageModule._generateGasMetadata()` sets the `_refundAddress` to `tx.origin`, a third-party-controlled address, `payForGas()` might call `tx.origin` with unbounded gas if an overpayment has occurred.

Prior to the Pectra hardfork, this was not an issue as `tx.origin` was always an EOA and could not reenter the system. However, it is now possible for `tx.origin` to have some code and reenter the system in an unexpected way.

Risk accepted:

Velodrome understands and accepts this risk.

5.2 Relayer Can Prevent Tokens From Being Deposited in the Escrow

Design | Low | Version 6 | Risk Accepted

CS-VELO-SC-INOP-021

In the `RootEscrowTokenBridge` contract, if a message of type `SEND_TOKEN_AND_LOCK_LENGTH` is received in the `handle()` function, the token bridge attempts to deposit the underlying tokens into the escrow contract.

```
else if (length == Commands.SEND_TOKEN_AND_LOCK_LENGTH) {
    (address recipient, uint256 amount, uint256 tokenId) = _message.sendTokenAndLockParams();
    IXERC20(xerc20).mint({_user: address(this), _amount: amount});

    IERC20(xerc20).safeIncreaseAllowance({spender: address(lockbox), value: amount});
    lockbox.withdraw({_amount: amount});
    erc20.safeIncreaseAllowance({spender: address(escrow), value: amount});
    try escrow.depositFor({_tokenId: tokenId, _value: amount}) {}
    catch {
        erc20.safeDecreaseAllowance({spender: address(escrow), requestedDecrease: amount});
        erc20.safeTransfer({to: recipient, value: amount});
    }
}
```

A malicious relayer can force the inner call to the escrow contract to fail by providing a carefully chosen gas limit. Although the inner call would run out of gas, given the 63/64 rule, the outer context might still have enough gas to execute the remaining logic in the `handle()` function, depending on the gas consumption of the `depositFor()` function.

This allows a malicious relayer to prevent `handle()` from depositing tokens into the escrow contract, causing the tokens to be sent back to the original recipient instead.

Risk accepted:

Velodrome understands and accepts this risk.

5.3 Sequencer Downtime Over Epoch Boundary

Security | Low | Version 2 | Acknowledged

CS-VELO-SC-INOP-014

With the introduction of root timestamps for `DEPOSIT` and `WITHDRAW` messages in Version 2 of the code, the system is exposed to a risk of sequencer downtime over an epoch boundary.

The blackout period at the end of an epoch and the distribution window period at the beginning of a new epoch allow the leaf chains to receive all messages from the root chain for the epoch that just ended and reach a consistent state after processing all of them.

However, due to sequencer downtime that lasts across the epoch boundary, the system can be exposed to the reordering of messages belonging to different epochs. This can lead to an inconsistent checkpoint state in the leaf gauge reward contract. Indeed, `Rewards.sol` relies on the assumption that deposit timestamps are monotonically increasing across epochs. While, reordering for a single epoch is handled correctly, receiving messages from epoch E and then E - 1 will create an additional checkpoint, breaking the assumption that when a message from epoch E is received, no messages from previous will be received later. Both the user's and supply checkpoint system can be affected. It is therefore crucial that the gauges are killed before the epoch flip in case of sequencer downtime that can last across an epoch boundary.

Acknowledged:

Velodrome acknowledges the risk and will update their incident response plan to include the above scenario.

5.4 `XERC20` Rate Limit Can Prevent Bridging of Funds

Design | Low | Version 1 | Risk Accepted

CS-VELO-SC-INOP-004

`XERC20` implements a rate limit for bridges for minting and burning operations. The rate limits are defined per bridge and thus can be different.

Situation can arise, where a `TokenBridge` has initiated a token bridging on a chain with high buffer cap. This Situation can arise naturally or artificially, for example, by a malicious actor. However, the destination bridge does not have enough buffer cap for the minting operation to succeed.

Choosing appropriate buffer cap is deciding a trade-off between security and usability.

Risk Accepted:

Velodrome accepts the risk.

6 Resolved Findings

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

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

Critical-Severity Findings	0
High-Severity Findings	2
Medium-Severity Findings	2
Low-Severity Findings	4
Informational Findings	7

- Voting Power Can Be Temporarily Artificially Inflated Code Corrected
- RootMessageBridge.sendMessage() Reverts if InterchainGasPaymaster Is Used Code Corrected
- Chain ID Must Be Mapped Code Corrected
- Voting Period in Epochs Can Be Bypassed by Using poke() Code Corrected
- Invalid Comparison of Domain With Chain ID Code Corrected
- Metadata Misuse in Bridges Code Corrected
- Killing and Reviving Leaf-Chain Gauges Code Corrected
- deployXERC20WithLockbox() in XERC20Factory Can Be Frontrun Code Corrected
- Gas Limit Discrepancy Code Corrected
- Arbitrary Address Can Be Revived as a Gauge Code Corrected
- Missing chainId Check in TokenBridge.handle() Code Corrected
- Specification Inaccuracy Specification Changed
- Usage of Both transferFrom() and safeTransferFrom() Code Corrected
- XERC20 Hardcoded 18 Decimals Specification Changed
- assert Statement in XERC20Factory Is Redundant Code Corrected

6.1 Voting Power Can Be Temporarily Artificially Inflated

Security High Version 2 Code Corrected

CS-VELO-SC-INOP-015

In Version 2 of the code, nonces for `WITHDRAW` and `DEPOSIT` messages were removed and replaced by timestamps. However, this introduces eventual consistency on the Leaf reward contract side as the order of the messages is not guaranteed. `GET_FEE` and `GET_BRIBES` can only be called in the next epoch if the user voted in the current epoch.

However, the following scenario is now possible:

- User `pokes()` at the end of epoch E on a `LeafGauge` G1.

- User claims rewards/bribes in epoch E+1 on G1.
- If the messages arrive at Leaf in the following order :

    1. `DEPOSIT` on either epoch `E` or `E+1`
    2. `GET_FEES/BRIBES` on epoch `E+1`
    3. `WITHDRAW` on epoch `E+1`

Then the user will have claimed rewards with inflated voting power. Note, that there is still a risk for the user to lose the rewards if the reordering of the messages is:

1. `WITHDRAW` on either epoch `E` or `E+1`
2. `GET_FEES/BRIBES` on epoch `E+1`
3. `DEPOSIT` on epoch `E+1`

Normal users can perform this with `poke()` while whitelisted users can perform this with `poke()` and `vote()`.

While in the above scenario, the user can inflate its reward by using `poke()` themself and claim rewards in the next epoch, it is also possible for a user that didn't vote in the epoch `E` to claim inflated rewards in the following scenario:

- Alice pokes/votes at the end of epoch E
- Bob who hasn't voted in this epoch claims rewards at the end of epoch E.
- If the messages arrive at Leaf in the following order in epoch E + 1 :

    1. Alice:WITHDRAW
    2. Bob:CLAIM_FEES
    3. Alice:DEPOSIT

    Bob will be able to claim rewards with inflated voting power as Alice's `DEPOSIT` message will be processed after Bob's `CLAIM_FEES` message.

    However, Bob can also lose rewards if the messages are reordered as follows:

    1. Alice:DEPOSIT
    2. Bob:CLAIM_FEES
    3. Alice:WITHDRAW

    In this case, Alice's voting power and the total votes will be inflated temporarily, reducing Bob's rewards.

Code corrected:

The code has been modified such that rewards cannot be claimed during the blackout period (the last hour before the epoch flip) and during the distribution window (the first hour after the epoch flip). This allows for enough time for the messages to be processed such that the reward contracts reach a coherent state before rewards can be claimed.

6.2 RootMessageBridge.sendMessage() Reverts if InterchainGasPaymaster Is Used

Design | High | Version 1 | Code Corrected

CS-VELO-SC-INOP-001

When `RootMessageBridge.sendMessage()` is called, first the fee is queried using `RootHLMessageModule.quote()`, then `RootHLMessageModule.sendMessage()` is called to send the message, with fee as msg.value.

This flow has two problems.

First, `RootHLMessageModule.quote()` will query `quoteDispatch()` without metadata. Default `InterchainGasPaymaster` hook will use default 50k gas limit to estimate the fee. However, during `RootHLMessageModule.sendMessage()` the gas metadata will be computed using `RootHLMessageModule._generateGasMetadata()` 50k gas is not enough, to fulfill any of the `GasLimits` contract limits. Thus, it will lead to the Hyperlane IGP hook reverting as the `msg.value` will always be smaller than the required `gasLimit` passed to when calling `Mailbox.dispatch()` call. Therefore, no message can be dispatched to the leaf chain.

Second, the `_messageBody` used `RootHLMessageModule.quote()` will be modified in the `RootHLMessageModule.sendMessage()` if `command <= Commands.WITHDRAW`. For those two commands (DEPOSIT and WITHDRAW), nonce is appended to the message body. We did not find any Hyperlane hook at this moment that relies on the content or length of the message body, however, modular nature of Hyperlane hooks might introduce such hook in the future.

Code corrected:

Changes were made to include the metadata in the gas quote. The nonce has been removed as part of another change, therefore the second issue is no longer relevant.

6.3 Chain ID Must Be Mapped

Correctness | Medium | Version 4 | Code Corrected

CS-VELO-SC-INOP-016

As the Hyperlane's domains do not necessarily match the chain ID, a mapping must be used to ensure the correctness of the data for the message being sent. In `RootMessageBridge.sendMessage()`, `_chainid` is used directly as `_destinationDomain` when requesting a quote. To cover the cases where `domain != chainID`, the mapping of `RootHLMessageModule` should be used to get the correct domain.

Code corrected: `RootHLMessageModule.quote()` now converts the chainID to domain ID.

6.4 Voting Period in Epochs Can Be Bypassed by Using poke()

Design | Medium | Version 1 | Code Corrected

CS-VELO-SC-INOP-002

Due to synchronization between the Root chain and the leaf chain, voting is disabled 1 hour before the end of the epoch and enable 1 hour into the epoch such that for example `DEPOSIT` messages emitted from the root chain have time to be processed on the leaf destination chain.

`Voter.vote()` allows a user to reassign his voting weight and `Voter.poke()` allows a use to revote the same weights as before.

However, while `Voter.vote()` only allows whitelisted NFT to vote during the voting blackout period, `Voter.poke()` allows any user to vote during it. Any user can `poke()` any other user. Thus, users are at risk of getting their rewards denied by malicious users who would `poke()` them near the end of the epoch if their votes are cast on leaf gauges and the cross-chain messages are processed on the leaf chain in the next epoch, but the withdrawals are processed in the current epoch.

Code corrected:

During the voting blackout period (1 hour prior to epoch flip, which occurs at 00:00 GMT time each Thursday), only the owner or approved user for a given NFT will now be able to `vote()` or `poke()`. This prevents other users from being able to poke a user maliciously prior to epoch flip.

6.5 Invalid Comparison of Domain With Chain ID

Design | Low | Version 6 | Code Corrected

CS-VELO-SC-INOP-018

In the `DomainRegistry` contract, the owner can call the `registerDomain` function to register a new Hyperlane domain. Functions inheriting from `DomainRegistry` use this mechanism to ensure that a remote domain is registered before sending a message to it or receiving messages from it.

```
function registerDomain(uint32 _domain) external onlyOwner {
    if (_domain == block.chainid) revert InvalidDomain();
    if (_domains.contains({value: _domain})) revert AlreadyRegistered();
    _domains.add({value: _domain});
    emit DomainRegistered({_domain: _domain});
}
```

As highlighted in the Hyperlane documentation, Hyperlane domains do not necessarily correspond to the chain ID, even for EVM-compatible chains. For instance, the domain for RARI Chain is `1000012617`, whereas the actual chain ID is `1380012617`. Consequently, the check `_domain == block.chainid` will not always prevent the current chain's domain from being registered.

Code corrected:

The `DomainRegistry` was updated and now compare the provided domain against `Mailbox(mailbox).localDomain()` instead of `block.chainid`.

6.6 Metadata Misuse in Bridges

Correctness | Low | Version 6 | Code Corrected

CS-VELO-SC-INOP-019

In `BaseTokenBridge`, the `_generateGasMetadata()` function is used to generate the metadata that is passed to the Hyperlane Mailbox along with the message to be sent.

```
function _generateGasMetadata(address _hook, uint256 _value, address _refundAddress, bytes memory _message)
    internal
    view
    virtual
    returns (bytes memory)
{
    /// @dev If custom hook is set, it should be used to estimate gas
    uint256 gasLimit = _hook == address(0) ? GAS_LIMIT() : IHookGasEstimator(_hook).estimateSendTokenGas();
    return StandardHookMetadata.formatMetadata({
        _msgValue: _value,
        _gasLimit: gasLimit,
        _refundAddress: _refundAddress,
        _customMetadata: ""
    });
}
```

Similarly, `LeafEscrowTokenBridge` and `RootHLMessageModule` respectively override and define the `_generateGasMetadata()` function in a comparable manner.

According to Hyperlane's documentation, the `_msgValue` field of the standard metadata is expected to represent the `msg.value` that will be sent along with the message on the destination chain by the relayer.

However, the metadata constructed and passed to `Mailbox.dispatch()` in `LeafTokenBridge`, `RootTokenBridge`, and `RootHLMessageModule` misuses this value. Instead of representing the `msg.value` for the destination chain, it provides the fee being paid on the source chain as `_msgValue` (i.e., `_msgValue` is set to the `msg.value` provided to `dispatch()`).

- In `LeafTokenBridge`, the metadata includes `msg.value` as the `_value` parameter. However, `msg.value` is also used to pay the bridge's fee.
- Similarly, in both `RootTokenBridge` and `RootHLMessageModule`, the metadata sets `_msgValue = msg.value` if the transaction is not sponsored. If the transaction is sponsored, `_msgValue` is set to `fee`, where `fee` represents the bridge fee covered by the `PaymasterVault` for whitelisted senders.

Note that if the hooks being used by the bridges do not rely on the `_msgValue` field of the metadata, this issue will be ignored by the Hyperlane system when quoting and charging fees.

Code corrected:

The `_generateGasMetadata()` functions were updated to hardcode the `_msgValue` field to `0` as the bridges are not expected to handle native token transfers.

6.7 Killing and Reviving Leaf-Chain Gauges

Design | Low | Version 1 | Code Corrected

CS-VELO-SC-INOP-003

`EmergencyCouncil.killRootGauge()` kills RootGauge only via the call to `Voter.killGauge()`. `EmergencyCouncil.killLeafGauge()` attempts to kill first the RootGauge via `Voter.killGauge()` and then the LeafGauge via a `KILL_GAUGE` message sent to the `RootMessageBridge`. If the RootGauge is already killed, the `Voter.killGauge()` call will revert with `GaugeAlreadyKilled` error. This design prevents killing a gauge on the leaf chain if it was already killed on the root chain.

The same holds for `EmergencyCouncil.reviveRootGauge()` and `EmergencyCouncil.reviveLeafGauge()`.

Code corrected:

The `kill` and `revive` functions for root gauges now revert if called on a leaf gauge. The check is done by checking for the presence of a chain id on the root contract.

6.8 `deployXERC20WithLockbox()` in `XERC20Factory` Can Be Frontrun

Security | Low | Version 1 | Code Corrected

CS-VELO-SC-INOP-005

`XERC20Factory.deployXERC20WithLockbox()` can be called by anyone. It deploys the `XERC20Lockbox` and the `XERC20` contract using `create3` function. The salt does not depend on `_erc20` parameter. Therefore, front-runing with a non-VELO `_erc20` parameter is possible. A redeployment of all `XERC20Factory` contracts on all chains and all `XERC20` contracts will be necessary to fix this issue.

Code corrected:

The `ERC20` token is now set in the constructor, which prevents the front-running from having an impact.

6.9 Gas Limit Discrepancy

Informational | Version 6 | Code Corrected

CS-VELO-SC-INOP-026

The `BaseTokenBridge` contract defines the `GAS_LIMIT()` function as follows:

```
function GAS_LIMIT() public pure virtual returns (uint256) {
    return 200_000;
}
```

The `RootTokenBridge` contract inherits from `BaseTokenBridge` but does not override the `GAS_LIMIT()` function.

The `RootEscrowTokenBridge` contract, which inherits from `RootTokenBridge`, overrides the `GAS_LIMIT()` function as follows:

```
function GAS_LIMIT() public pure override(BaseTokenBridge, ITokenBridge) returns (uint256) {
    return 76_000;
}
```

However, since the `LeafEscrowTokenBridge` contract does not override the `handle()` function defined in `LeafTokenBridge`, messages processed by both `LeafEscrowTokenBridge` and `LeafTokenBridge` should consume a comparable amount of gas.

Additionally, note that the `LeafEscrowTokenBridge` contract overrides the `GAS_LIMIT()` function to return `190_000`, whereas the `LeafTokenBridge` contract inherits a `GAS_LIMIT()` of `200_000` from `BaseTokenBridge`. This discrepancy suggests that messages processed by the `RootTokenBridge` may consume more gas than the `RootEscrowTokenBridge`, despite the code hinting the opposite.

Code corrected:

The `RootEscrowTokenBridge` no longer overrides the `GAS_LIMIT()` function. Moreover, the `BaseTokenBridge` contract now defines the `GAS_LIMIT()` function to return `76_000` gas. If a bridge requires a higher value, it will need to override the function, which done by the `LeafEscrowTokenBridge` with `190_000` gas.

6.10 Arbitrary Address Can Be Revived as a Gauge

Informational | Version 1 | Code Corrected

CS-VELO-SC-INOP-006

In `EmergencyCouncil`, `reviveRootGauge` can be called on an arbitrary address that is not registered as a gauge. Indeed, `Voter.reviveGauge()` will also not verify that the provided address is a registered gauge and update the `isAlive` mapping accordingly. This can lead to a situation where an arbitrary address can be considered as an alive gauge from an external perspective.

This behavior differs from `reviveLeafGauge` which will in `LeafVoter` check that the provided address is indeed a registered gauge through the `isGauge` mapping.

Code corrected:

Both `reviveRootGauge` and `reviveLeafGauge` now check that the provided address is a registered gauge with `Voter.isGauge()`.

6.11 Missing `chainId` Check in `TokenBridge.handle()`

Informational | Version 1 | Code Corrected

CS-VELO-SC-INOP-017

In `TokenBridge.handle()` there is no verification that the cross-chain message originated from a registered chain. Therefore, a small risk exists that a malicious actor manages to deploy a contract at the same address as the `TokenBridge` contract on a different chain which could then send a message to the `TokenBridge` contract on the main chain to mint `xVELO` tokens. However, this requires the attacker to deploy a contract at a specific address which is highly unlikely.

Code corrected:

A check was added to ensure that only messages from registered chains are processed.

6.12 Specification Inaccuracy

Informational | Version 1 | Specification Changed

CS-VELO-SC-INOP-008

In the specification file `SPECIFICATION.md` the following description is given for the format of `GET_INCENTIVES` and `GET_FEES` messages :

```
##### Get Incentives & Get Fees
This payload consists of:

- 1 byte command
- 20 byte gauge address
- 32 byte recipient address
- 32 byte token id
- 1 byte tokens array length
- 20 - 100 bytes of token addresses
```

However, the `recipient address` is not 32 byte long but a 20 byte address. Furthermore, the `token addresses` are not 20-100 bytes long but can vary from 0 to 160 bytes (5 * 32) as elements of an array that is `encodedPacked` are still padded according to the specification

---

Specification changed:

The specification have been updated to reflect the correct length of the recipient address and the token addresses.

6.13 Usage of Both `transferFrom()` and `safeTransferFrom()`

Informational | Version 1 | Code Corrected

CS-VELO-SC-INOP-009

`Voter.notifyRewardAmount()` uses `safeTransferFrom()` with `rewardToken`. `Voter.distribute()` first, calls `rewardToken` with `safeApprove()` and then e.g. `RootGauge.notifyRewardAmount()`. `RootGauge.notifyRewardAmount()` uses `transferFrom()` with `rewardToken`. As the reward token is assumed to be `VELO` this is not immediately an issue. However, the use of `transferFrom()` restricts which tokens can be used as reward token.

Code corrected:

Code now uses `safeTransferFrom()` in `RootGauge.notifyRewardAmount()` and `RootGauge._notify()` instead of `transferFrom()`.

6.14 `XERC20` Hardcoded 18 Decimals

Informational | Version 1 | Specification Changed

CS-VELO-SC-INOP-012

`XERC20` decimals are fixed to 18. However, the `XERC20Lockbox` can be created and mint 1:1 any arbitrary ERC20 token. `XERC20Lockbox` and `XERC20Factory` don't enforce that the decimals of the wrapped ERC20 token are 18. It is therefore assumed that `XERC20` is only used to wrap `ERC20` tokens with 18 decimals, such as `VELO`.

Specification changed:

Velodrome acknowledges the issue and updated the specification in the `XERC20Factory` to make support for 18 decimals explicit.

6.15 `assert` Statement in `XERC20Factory` Is Redundant

Informational **Version 1** **Code Corrected**

CS-VELO-SC-INOP-013

In `XERC20Factory.deployXERC20WithLockbox()`, the following assertion is performed after `deployCreate3()` has been invoked to deploy both the lockbox and the `XERC20` contract: `assert(_XERC20 == expectedAddress);`. As `expectedAddress` is computed using `CREATEX` with the same parameters as in `deployCreate3()`, the assertion is redundant and can be removed.

Code corrected:

The `assert` statement has been removed.

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 Gas Savings

Informational **Version 6** **Acknowledged**

CS-VELO-SC-INOP-022

In the `DomainRegistry` contract, both the `registerDomain` and `deregisterDomain` functions include logic to check whether a domain is already registered or not, reverting if the condition is met.

```
function registerDomain(uint32 _domain) external onlyOwner {
    if (_domain == block.chainid) revert InvalidDomain();
    if (_domains.contains({value: _domain})) revert AlreadyRegistered();
    _domains.add({value: _domain});
    emit DomainRegistered({_domain: _domain});
}
```

Since `_domains.add()` and `_domains.remove()` already handle failure cases by returning false (e.g `add()` return false the domain is already registered), the explicit `contains()` check could be omitted and instead, the logic could rely on the return values of `add()` and `remove()` to save the gas consumed by the storage loads made in `contains()`.

Acknowledged:

Velodrome acknowledges this informational issue and decided not to apply the recommendation given the low impact of the gas savings.

7.2 Locked Funds

Informational **Version 6** **Acknowledged**

CS-VELO-SC-INOP-023

Funds can become locked in the various bridge contracts defined in the system, with no mechanism to unlock them.

In the different `TokenBridge` contracts, the `handle()` function is marked as payable. However, there is no implemented method to withdraw native tokens from the contract.

Similarly, there is no function to recover ERC20 tokens that might be sent to the contract by mistake.

Acknowledged:

Velodrome acknowledges this informational issue.

7.3 Missing Sanity Checks

Informational | Version 6 | Acknowledged

CS-VELO-SC-INOP-024

In `BaseTokenBridge.constructor()`, the different address being provided as parameters are not sanity checked. It could be useful to check that the addresses are not zero addresses.

Acknowledged:

Velodrome acknowledges this informational issue.

7.4 `SentMessage` Event Inaccurately Reflects the Fee Paid

Informational | Version 6 | Acknowledged

CS-VELO-SC-INOP-025

In the `LeafTokenBridge` and `RootTokenBridge` contracts, the event `SendMessage` is emitted when a message is sent.

Before Version 6, this event displayed the exact fee paid for bridging the message. However, it now only shows the message value provided. The actual fee paid might be smaller if a refund occurred in Hyperlane.

Acknowledged:

Velodrome acknowledges this informational issue and added:

```
Hyperlane fees are not regularly updated, so this is unlikely to be a material change
```

7.5 Potential Reentrancy Due to `InterchainGasPaymaster` Calling `tx.origin`

Informational | Version 1 | Risk Accepted

CS-VELO-SC-INOP-007

Hyperlane `Mailbox` might have `InterchainGasPaymaster` as one of its default hooks. `InterchainGasPaymaster.payForGas()` is triggered as a post-dispatch hook. `RootHLMessageModule._generateGasMetadata()` sets the `_refundAddress` to `tx.origin` - 3rd party controlled address. In `InterchainGasPaymaster.payForGas()`, might call `tx.origin` with unbounded gas if overpayment has occurred.

The refund should never happen in practice, as the `RootMessageBridge.sendMessage()` aims at transferring the exact amount of gas defined by the `gasLimit` from the user to hyperlane.

However, this case must be considered as a potential reentrancy vector.

Risk accepted:

Velodrome acknowledges the risk.

7.6 Whitelisted NFTs Can Double Voting Power

Informational | Version 1 | Risk Accepted

CS-VELO-SC-INOP-010

Whitelisted NFTS in `Voter` can `vote()` during the voting blackout period. Following sequence of events is possible:

- Root chain at Epoch E: Alice votes on a leaf gauge deployed on chain C1
- Root chain at the last moment of Epoch E+1: Alice votes on another leaf gauge on chain C2
- Leaf chain C2 at Epoch E+1: The deposit message from the vote for the gauge on chain C2 is processed
- Leaf chain C1 at Epoch E+2: The withdrawal message from the vote for the gauge on chain C1 is processed

The order of DEPOSIT and WITHDRAW messages is not guaranteed between leaf chains. If this situation arises, Alice will be able to claim rewards for epoch E+1 on both C1 and C2 chains.

Risk Accepted:

Velodrome acknowledges the risk and accepts it as is.

7.7 `LeafGauge._claimFees()` Return Values Are Never Used

Informational | Version 1 | Acknowledged

CS-VELO-SC-INOP-011

In `LeafGauge._claimFees()`, the return value is not used by `LeafGauge.notifyRewardAmount()` - the only place where this it is called.

Acknowledged:

Velodrome acknowledges the issue but does not plan to change the code.

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 Bribes Can Be Added After Voting Is Over

Note Version 1

Voting is only available 1 hour into the epoch and closes 1 hour before the epoch ends. This is cooldown period serves as buffer to ensure that Withdrawal and Deposit actions has been finalized on the leaf chain. However, bribes can be added at any time to a `BribeVotingReward` contract using `notifyRewardAmount()`. Users will not be able to react to these bribes after the voting period is over. Distributing bribes with significant time left in the epoch is encouraged to allow users to react to the bribes.

8.2 Bridging From Leaves

Note Version 1

When bridging from the Root chain to a Leaf, the target chain ID is translated into the corresponding Hyperlane domain ID.

However, when bridging from a Leaf, no such translation occurs. Currently, the only whitelisted chain is the Root chain, which does not require translation. If a new whitelisted chain were added that does require translation, bridging from a Leaf to this chain would fail.

8.3 ChainRegistry Contracts Can Be Misconfigured

Note Version 1

If chain `A` can send messages to chain `B`, `ChainRegistry` contract on chain `A` must have chain `B` registered by the owner. However, there is no guarantee that chain `B` has chain `A` registered in his `ChainRegistry` contract. Thus, a situation can occur where `A` sends message to `B` but `B` cannot send back to A.