PUBLIC

Code Assessment

of the Epoch Governor Smart Contracts

January 13, 2025

Produced for

by

Contents

1 Executive Summary

Dear Velodrome team,

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

Velodrome implements two governance mechanisms to direct the emission rate of the VELO token. The SimpleEpochGovernor allows a trusted EOA or MultiSig to change the emission rate, and the EpochGovernor implements a system where stakers of VELO in the Velodrome protocol can vote on how to change the emission rate.

The most critical subjects covered in our audit are proposal execution correctness, proposal sanitization during creation, and signature handling. Issues reported in the first version of the code were satisfactorily addressed. Security regarding all aforementioned topics is high.

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	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	7
• Code Corrected	4
• Specification Changed	1
• Risk Accepted	1
• 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 Epoch Governor 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	25 November 2024	a2e1ec9d1e720e7c79416cc19f393d2d31c337f2	Initial Version
2	13 December 2024	bfd33ad4c8630be5e82a28b0f31820c6273944a5	Second version
3	20 December 2024	e8e91ccf2fa90d29901576131060a0b0e3b5691b	fixed voting start

For the solidity smart contracts, the compiler version `0.8.25` was chosen.

The following files are considered in scope for this assessment:

```
contracts/governance/EpochGovernorCountingFractional.sol
contracts/governance/GovernorCommentable.sol
contracts/governance/GovernorProposalWindow.sol
contracts/governance/GovernorSimple.sol
contracts/governance/GovernorSimpleVotes.sol
contracts/EpochGovernor.sol
contracts/SimpleEpochGovernor.sol
```

2.1.1 Excluded from scope

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

2.2 System Overview

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

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

Velodrome offers a governance voting system that allows users to propose and vote on the tail emission rate of the VELO token.

The Velodrome protocol operates in epoch, where each epoch is 1 week long. Each epoch, VELO tokens are minted and distributed to LP stakers depending on the amount of votes each gauge received, and to VELO stakers to compensate them against dilution. Emissions start initially at 15M tokens per epoch with a decay rate of 1% per epoch. Once the emission rate reaches 6M tokens per epoch (after 92 epochs) the weekly emissions enter a tail regime where they become a percentage of the token's total supply.

Initially this percentage is set to 30 basis points, however it can be increased by 1 bps, decreased by 1 bps, or left unchanged by a governance vote each epoch. The tail emission rate can't exceed 100 bps.

2.2.1 SimpleEpochGovernor

The `SimpleEpochGovernor` contract provides a basic governance mechanism for the Velodrome protocol. The contract allows the `governor` role, defined within the `Voter` contract, to `setResult()` of the tail emission rate update decision. The governor can then execute the decision using `executeNudge()`, which calls `Minter.nudge()` to apply the result of the decision to the tail emission rate of the VELO token. The `Minter` calls back into `SimpleEpochGovernor.result()` to query the intended action.

The result of the decision can either be : `Succeeded`, `Defeated` or `Expired`. Which respectively correspond to increasing by 1 bps, decreasing by 1 bps, or leaving the tail emission rate unchanged.

2.2.2 EpochGovernor

The `EpochGovernor` contract offers a more advanced governance framework, utilizing a forked version of OpenZeppelin governance. It allows users to propose and vote on the tail emission rate using their veVELO tokens as voting power. Every epoch, a proposal for the emission rate update is created, voted, and executed.

Users of Velodrome can lock their VELO tokens for a specific amount of time in the `VotingEscrow` contract to receive a NFT whose balance represents the voting weight of the escrowed tokens. The voting weight is calculated based on the amount of VELO tokens locked and the duration of the lock and decays linearly over time, from a maximum of four years. An NFT is represented by a `tokenId`.

2.2.2.1 Proposal Creation

A proposal can be created by the owner of the `EpochGovernor` contract within the first 24 hours of the epoch through `propose()`. If no proposal is made within this time, anyone can create a proposal until the end of the epoch. A proposal consists of a list of `_targets` to call with `_values` and `_calldatas`. In this case, `EpochGovernor._propose()` constrains these values to only allow the `Minter` contract as the single target and the `nudge()` function as calldata.

A proposal can only be created if another proposal with the same `proposalId` is not already existing. Since `proposalId` is a function of the epoch in which the proposal is created, the amount of new proposals is limited to one per epoch. A proposal always consists of three options to vote on: `AGAINST`, `FOR` and `ABSTAIN`.

2.2.2.2 Proposal States

A proposal can have multiple states in the system :

- *Pending* : The proposal has been created but the voting period has not yet started.
- *Active* : The proposal is open for voting.
- *Succeeded* : The proposal has passed with a majority vote of `FOR`, determined after the voting period has ended.
- *Defeated* : The proposal has been rejected with a majority vote of `AGAINST`, determined after the voting period has ended.
- *Expired* : The proposal has been resolved with a majority vote of `ABSTAIN`, determined after the voting period has ended.
- *Executed* : The proposal has been successfully executed.
- *Cancelled* : The proposal has been cancelled. This state is not reachable, as cancellations are not implemented.

- *Queued* : The proposal has been queued for execution. This state is not reachable as the queuing mechanism is disabled.

2.2.2.3 Voting

Once a proposal is in the *Active* state, when the current timestamp is between the proposal snapshot (exclusive) and the voting end (inclusive), users can vote on it. The voting power of a user is determined by the amount of veVELO held by the veVELO NFT they own. A user can own multiple veVELO NFTs, in which case he can vote with each one of them. The voting power of a user is queried from the historical balances of the voting escrow, at the timestamp defined as the "proposal snapshot", the time of the next block after the proposal creation or 1 hour after the epoch start (whichever comes later). Only the owner of the NFT at the snapshot time can vote with its assigned voting power, and since the snapshot is in the past, the NFT can't be transferred and its voting power reused. A proposal is active from the next block after the snapshot and remains active for the duration of the voting period.

Users can vote by directly calling the following functions in `SimpleGovernor` : `castVote()`, `castVoteWithReason()`, `castVoteWithReasonsAndParams()`, or by generating a signature that can be executed by a third party using `castVoteBySig()` or `castVoteWithReasonAndParamsBySig()`.

Votes can either be cast for a specific outcome, or fractional voting can be used. In fractional voting, the user can specify how much of their voting weight each option should receive.

A `reason` can be specified by the user when voting to provide additional context to the decision made.

2.2.2.4 Proposal Execution

After the voting period has ended, the proposal will have one for the three following states : `Succeeded` representing an increase in tail emission rate, `Defeated`, representing a decrease, or `Expired`, representing no change. However, if there is a tie between two majority outcomes, the proposal will be marked as `Expired` indicating that the tail emission rate will be left unchanged.

The proposal can be executed by anyone during the hour that follows the end of the voting period. If the proposal is not executed between the voting period end and the epoch end, it is impossible to ever execute it. Not executing the proposal will result in the tail emission rate remaining unchanged.

2.2.2.5 Contract Inheritance and Functionality

`EpochGovernor` inherits from the following contracts:

- `GovernorProposalWindow`: Implements the logic to restrict proposal creation during the first `proposalWindow` hours of the epoch, after which any user can create a proposal if one has not been created yet. By default, the `proposalWindow` is set to 24 hours.
- `GovernorCommentable`: Allows users holding a sufficient fraction of the voting power to `comment()` on proposals. The required fraction is configurable by the contract owner through `setCommentWeighting()`, with the default threshold being 0.0004% of the total veVELO supply.
- `GovernorSimpleVotes`: Retrieves the historical voting power of veNFTs owned by users at the snapshot time.
- `GovernorSimple`: A modified version of OpenZeppelin's `Governor` contract. Key modifications include the removal of the `cancel()` function and the introduction of a `tokenId` argument representing a veNFT when casting a vote.
- `EpochGovernorCountingFractional`: Implements vote counting in `_countVote()`, allowing users to assign voting power in different ways. Additionally, it implements the `_selectWinner()` logic to determine the outcome of a proposal.

2.2.3 Roles & Trust Model

`EpochGovernor` has an `owner` that can set the `proposalWindow` parameter (limited to 24 hours max), and the `commentingWeighting` parameter. A malicious owner cannot tamper with the voting process and result. `SimpleEpochGovernor` is controlled by `Voter.governor()` which is fully trusted.

It is assumed that:

- proposal creation happens before the last hour of an epoch
- proposals are executed before the end of the epoch.
- `updatePeriod()` has already been called on `Minter` for the current epoch before a proposal is executed.

2.2.4 Changes in Version 2

The computation of the `proposalId` has been modified such that the result is a hash of all proposal parameters (target address, calldata and value) and the time of the end of voting for the current epoch. On proposal creation, it is enforced that the proposal has a single target, equal to the `Minter`, calldata equal to the `nudge()` function selector, and a value of `0`.

The signatures used to vote with `castVoteBySig()` and `castVoteWithReasonAndParamsBySig()` have been modified to include the `tokenId` in the signed data.

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:

- 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	2

5.1 EpochGovernor Should Not Receive ETH or Other Tokens

Design Low Version 1 Risk Accepted

CS-VELOGOV-004

`EpochGovernor` derives from `GovernorSimple` which exposes a non reverting `receive()` function, and methods `onERC721Received()`, `onERC1155Received()`, and `onERC1155BatchReceived()` which allow `EpochGovernor` to receive ETH, and ERC1155 and ERC721 tokens through the `safeTransferFrom()` functions.

There is no way to transfer out ETH or tokens that have been received by `EpochGovernor`, so there is no reason to expose functions to accept tokens.

Risk accepted:

Velodrome accepts the risk with the following statement:

"These changes will not prevent ERC20s from being transferred in. The risk for such events are low as well, given that these contracts will mainly be interacted with through a UI."

5.2 Voting Power Queried by `comment()` Could Be in Future

Design Low Version 1 Acknowledged

CS-VELOGOV-009

The `comment()` method requires the proposal to be in the `Pending` or `Active` state. If the proposal is in the `Pending` state, the proposal snapshot is in the future or present, and the voting power queried through `_getVotes()` is not finalized yet. A user could use the voting power of a given `tokenId` to create a comment, and then transfer the `tokenId` for its voting power to be reused by another user.

Since the only effect of commenting is event emission, there are no adverse consequences.

Acknowledged:

Velodrome acknowledges the issue with the following statement:

> The voting power restriction's primary use case is as a spam filter, so we acknowledge the risk. For the `EpochGovernor`, it is unlikely to make a huge difference given that the maximum time between proposal creation and the vote starting is less than an hour.

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	1
Medium-Severity Findings	2
Low-Severity Findings	5
Informational Findings	4

- Arbitrary Payload in execute() Code Corrected

- Missing Checks in propose() Code Corrected
- Signature Does Not Include tokenId Code Corrected

- Incorrect supportsInterface() Result Code Corrected
- Vote Duration Is Two Seconds Too Long Code Corrected
- _countVote() Can Revert With Incorrect Error Message Code Corrected
- getVotes() Uses Past Timestamp With Current Owner Specification Changed
- votingPeriod() Returns Incorrect Value Code Corrected

- Incorrect NatSpec for _castVote() Specification Changed
- Missing Events Code Corrected
- Unused Imports and Variables Code Corrected
- Voting Escrow Public View Exposed Multiple Times Code Corrected

6.1 Arbitrary Payload in `execute()`

Security High Version 1 Code Corrected

CS-VELOGOV-001

The `execute()` method of `EpochGovernor` accepts `_targets`, `_values`, and `_calldatas` as parameters, which are supposed the same as the one specified during proposal creation through `propose()`. However, this is not enforced, and the `execute()` arguments can be arbitrarily specified by the untrusted caller.

Generally, the `execute()` function of OpenZeppelin Governance module hashes all arguments to produce the `proposalId`, so the `proposalId` is linked to the proposal payload. However, `EpochGovernor` only hashes the timestamp of the vote end to obtain the `proposalId`, so the `execute()` payload is not validated implicitly by the hashing function.

`EpochGovernor` is not expected to have special privileges in the overall Velodrome/Aerodrome systems except for the `nudge()` function of *Minter*, so arbitrary calls are not expected to pose a critical threat. However, a malicious caller controlling the payload of `execute()` would be able to mark the proposal as executed without calling `Minter.nudge()`.

Code corrected:

`hashProposal()` is no longer overridden in `EpochGovernor`. Therefore, the proposal parameters are now taken into account when calculating the proposal hash. This ensures that the proposal can only be executed with the parameters specified during proposal creation.

6.2 Missing Checks in `propose()`

Design | Medium | Version 1 | Code Corrected

CS-VELOGOV-002

The arguments of `EpochGovernor.propose()` can be misused in multiple ways to create invalid proposals for which execution will fail:

- The `_values` array, which specifies the ETH values transferred with every call to `_targets`, is not enforced to only contain zero values. As a consequence, calls to `Minter.nudge()` with a non-zero value will fail because `nudge()` is non-payable, and `EpochGovernor` does not hold ETH.
- `_calldatas[0]` is enforced to start with the 4 bytes of the `nudge()` selector, but there is no bound to the maximum length of the calldata. A malicious proposer creator can specify a very long calldata string, such that the gas cost of calling `execute()` becomes problematic.

Code corrected:

The `propose()` function has been modified to check that the `_values` array only contains zero values. The function now also checks that the calldata only contains the 4 bytes of the `nudge()` selector.

6.3 Signature Does Not Include `tokenId`

Design | Medium | Version 1 | Code Corrected

CS-VELOGOV-003

The digest signed for methods `castVoteBySig()` and `castVoteWithReasonAndParamsBySig()` does not include the `_tokenId` argument. This means that a user's signature can be used with any `tokenId` owned by the user.

This could be exploited by a malicious actor to invalidate signatures of legitimate users:

1. Donate `tokenId` with a dust amount of veVELO balance to victim.
2. Acquire signature of victim.
3. Use signature of victim to cast vote with dust `tokenId`.

The risk is significantly mitigated by the sequencer being centralized on Optimism/Base and not having a public mempool, where signatures could be acquired and front-run.

Code corrected:

The `_tokenId` argument is now included in the digests signed for methods `castVoteBySig()` and `castVoteWithReasonAndParamsBySig()`.

6.4 Incorrect `supportsInterface()` Result

Design | Low | Version 1 | Code Corrected

CS-VELOGOV-005

Method `supportsInterface()` of `GovernorSimple` returns true if argument `_interfaceId` is equal to `type(IGovernor).interfaceId ^ IOZGovernor.cancel.selector`.

Since `IGovernor` does not include the `cancel()` method, XORing `IGovernor.interfaceId` with `cancel.selector` amounts to defining an `interfaceId` that includes all methods of `IGovernor` plus `cancel`. However, `cancel` is not implemented in `GovernorSimple`, so the `interfaceId` should not include it.

Code corrected:

`supportsInterface()` has been modified to not include `cancel.selector` in the `interfaceId`.

6.5 Vote Duration Is Two Seconds Too Long

Correctness | Low | Version 1 | Code Corrected

CS-VELOGOV-008

In `EpochGovernor._propose()`, the snapshot timestamp (`proposal.voteStart`) is computed by adding `votingDelay()` to the `voteStart` variable, so that it is at least two seconds in the future:

```
uint256 voteStart = Math.max({a: clock(), b: VelodromeTimeLibrary.epochVoteStart({timestamp: block.timestamp})});
proposal.voteStart = SafeCast.toUint48({value: voteStart + votingDelay()});
proposal.voteDuration = SafeCast.toUint32(epochVoteEnd - voteStart);
```

`proposal.voteDuration` is computed so that the voting should end exactly at `epochVoteEnd`. However, in

```
proposal.voteDuration = SafeCast.toUint32(epochVoteEnd - voteStart);
```

`voteStart` is used instead of `proposal.voteStart`. `voteStart` is 2 seconds less than `proposal.voteStart`. As a result, proposals end at `epochVoteEnd + 2`.

Code corrected:

In Version 3, `voteDuration` in `_propose()` is now correctly computed as `epochVoteEnd - voteStart`, and `voteStart` is now equal to `proposal.voteStart`.

6.6 `_countVote()` Can Revert With Incorrect Error Message

Design | Low | Version 1 | Code Corrected

CS-VELOGOV-010

`_countVote()`, defined in `EpochGovernorCountingFractional`, reverts with error message `GovernorAlreadyCastVote` if `remainingWeight` is 0. However, `remainingWeight == 0` could also mean that the user voting does not own the given `_tokenId`, so that `_totalWeight == 0`.

In this case, the revert is not happening because the vote has already been cast, like the error message would suggest, but because the user does not have voting power for the given `_tokenId`.

Code corrected:

`_castVote()` now reverts with error `GovernorZeroVotingPower` if the user has 0 voting weight (they do not own the `_tokenId`). The internal function `_countVote()` is no longer reached if the user has 0 voting weight.

6.7 `getVotes()` Uses Past Timestamp With Current Owner

Correctness | Low | Version 1 | Specification Changed

CS-VELOGOV-006

External view method `getVotes()`, defined in `GovernorSimpleVotes` should return the voting power of `_tokenId` at the given `_timepoint`. It is defined as follows:

```
function getVotes(uint256 _tokenId, uint256 _timepoint) external view returns (uint256) {
    address account = ve.ownerOf({tokenId: _tokenId});
    return _getVotes(account, _tokenId, _timepoint, "");
}
```

If `_timepoint` is in the past, the statement `account = ve.ownerOf(_tokenId)` is incorrect, as `ownerOf()` returns the current owner, not the owner at `_timepoint`. Since the current owner and the past owner could differ, the following `_getVotes(account, _tokenId, _timepoint, "")` could incorrectly return `0`, if in the past `_tokenId` was owned by another account.

Specification changed:

The documentation for `getVotes()` has been modified to include information on expected behavior when the user does not own the NFT. This function is currently only used for tests, and should not be used by integrators unless this quirk in behavior is fully understood.

6.8 `votingPeriod()` Returns Incorrect Value

Design | Low | Version 1 | Code Corrected

CS-VELOGOV-007

View method `votingPeriod()` is specified to be the time between vote start and vote end in `IGovernor.sol`:

```
/**
* @dev Delay between the vote start and vote end. The unit this duration is expressed in depends on the clock
* (see ERC-6372) this contract uses.
*/
```

In `EpochGovernor` it is implemented to return `1 week`, however the voting time is at most `1 week - (2 hours and 2 seconds)`, since the voting starts at earliest 1 hour and 4 seconds after the epoch start, and finishes (inclusive) 1 hour before the epoch end.

Code corrected:

`votingPeriod()` has been modified to return the maximum possible length of time a vote can be active.

6.9 Incorrect NatSpec for `_castVote()`

Informational | Version 1 | Specification Changed

CS-VELOGOV-011

The NatSpec of `SimpleGovernor._castVote()` says:

```
``Internal vote casting mechanism: Check that the vote is pending,``.
```

However, the function requires that the proposal is `Active`, not `Pending`.

Specification changed:

The documentation has been modified to reflect the correct behavior of the function.

6.10 Missing Events

Informational | Version 1 | Code Corrected

CS-VELOGOV-013

In `GovernorCommentable`, event `SetCommentWeighting` is not emitted when the commenting weight is initalized to `4000` in the (implicit) constructor.

In `GovernorProposalWindow`, event `ProposalWindowSet` is not emitted when `proposalWindow` is set to `24 hours` in the (implicit) constructor.

Events `SetCommentWeighting` and `ProposalWindowSet` also do not share the same naming style.

Code corrected:

The events are now emitted in the constructors of the respective contracts and the naming style has been corrected.

6.11 Unused Imports and Variables

Informational | Version 1 | Code Corrected

CS-VELOGOV-015

1. `IVotingEscrow` is not used in `EpochGovernor`.
2. The library `DelegationHelperLibrary` in `EpochGovernor` is unused. No variable of type IVotingEscrow exist in `EpochGovernor`. The following statement has no use:

   ```
   using DelegationHelperLibrary for IVotingEscrow;
   ```

3. Private constant `ALL_PROPOSAL_STATES_BITMAP` is unused in `GovernorSimple`. Since it is private it cannot be used in a derived contract either.

4. The constructor of `GovernorCommentable` accepts `_voter` as a parameter, but it is only used to query the voting escrow address. The voting escrow could be passed directly instead.

Code corrected:

The unused imports and variables have been removed.

6.12 Voting Escrow Public View Exposed Multiple Times

Informational | Version 1 | Code Corrected

CS-VELOGOV-016

The Voting Escrow address is exposed multiple times in `EpochGovernor` with different getters:

1. `ve()`, implemented in `GovernorSimpleVotes`.
2. `token()`, implemented in `GovernorSimpleVotes`.
3. `escrow()`, implemented from `GovernorCommentable`.

Code corrected:

`GovernorSimple` now stores the address of the voting escrow in a single variable, `ve`. This allows it to be used in all contracts derived from `GovernorSimple`.

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 Unreachable Code

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

CS-VELOGOV-014

Some code and conditions are unreachable:

1. lines 68-70 of `EpochGovernor.sol`: a proposal cannot be cancelled, so the following block cannot be entered.

```
if (proposalCanceled) {
    return ProposalState.Canceled;
}
```

2. lines 206-213 and 224-226 of `EpochGovernor.sol` in `execute()`. `_executor()` is hard-coded to `address(this)`, so the conditions `_executor() != address(this)` are never true. Moreover, the second condition `_targets[i] == address(this)` can not be satisfied since `_targets` can only contain the `Minter` address.

Acknowledged:

Velodrome acknowledges that the code is unreachable and states that there will be no changes.