Velodrome Finance contest Findings & Analysis Report

2022-08-08

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Velodrome Finance smart contract system written in Solidity. The audit contest took place between May 23—May 30 2022.

Wardens

82 Wardens contributed reports to the Velodrome Finance contest:

1. xiaoming90
2. Ruhum
3. smiling_heretic
4. WatchPug (jtp and ming)
5. lllllll
6. hyh
7. unforgiven
8. rotcivegaf
9. kenzo
10. 0x1f8b
11. hansfriese
12. cccz
13. Chom
14. codexploder
15. 0x52
16. Picodes
17. 0xNazgul
18. csanuragjain
19. p_crypt0
20. hake
21. MiloTruck
22. Dravee
23. MaratCerby
24. minhquanym
25. sashik_eth
26. horsefacts
27. pauliax
28. cryptphi
29. Funen

30. gzeon
31. berndartmueller
32. _Adam
33. GimelSec (rayn and sces60107)
34. simon135
35. c3phas
36. teddav
37. TerrierLover
38. djxploit
39. catchup
40. asutorufos
41. delfin454000
42. oyc_109
43. sach1r0
44. fatherOfBlocks
45. reassor
46. 0xf15ers (remora and twojoy)
47. 0xNineDec
48. Hawkeye (0xwags and 0xmint)
49. robee
50. Nethermind
51. jayjonah8
52. sorrynotsorry
53. BouSalman
54. AlleyCat
55. CertoraInc (egjlmn1, OriDabush, ItayG, and shakedwinder)
56. RoiEvenHaim
57. SooYa
58. supernova

59. 0x4non
60. Waze
61. saian
62. MadWookie
63. 0xkatana
64. UnusualTurtle
65. ElKu
66. DavidGialdi
67. rfa
68. Tomio
69. TomJ
70. z3s
71. Deivitto
72. Fitraldys
73. orion
74. ControlCplusControlV
75. Randyyy

This contest was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 23 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 17 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 50 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 51 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Velodrome Finance contest repository, and is composed of 17 smart contracts written in the Solidity programming language and includes 3,443 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (6)

[H-01] Users can get unlimited votes

Submitted by lllllll, also found by kenzo

Users can get unlimited votes which leads to them:

1. gaining control over governance
2. getting undeserved rewards
3. having their pools favored due to gauge values

Proof of Concept

`_mint()` calls `_moveTokenDelegates()` to set up delegation...

File: contracts/contracts/VotingEscrow.sol #1

```
462       function _mint(address _to, uint _tokenId) internal re
463           // Throws if `_to` is zero address
464           assert(_to != address(0));
465           // TODO add delegates
466           // checkpoint for gov
467           _moveTokenDelegates(address(0), delegates(_to), _t
```

VotingEscrow.sol#L462-L467

and `_transferFrom()` calls `_moveTokenDelegates()` to transfer delegates...

File: contracts/contracts/VotingEscrow.sol #2

```
301       function _transferFrom(
302           address _from,
303           address _to,
304           uint _tokenId,
305           address _sender
306       ) internal {
307           require(attachments[_tokenId] == 0 && !voted[_toke
308           // Check requirements
309           require(_isApprovedOrOwner(_sender, _tokenId));
310           // Clear approval. Throws if `_from` is not the cu
311           _clearApproval(_from, _tokenId);
312           // Remove NFT. Throws if `_tokenId` is not a valic
313           _removeTokenFrom(_from, _tokenId);
314           // TODO delegates
315           // auto re-delegate
316           _moveTokenDelegates(delegates(_from), delegates(_t
```

VotingEscrow.sol#L301-L316

but `_burn()` does not transfer them back to `address(0)`

File: contracts/contracts/VotingEscrow.sol #3

```
517       function _burn(uint _tokenId) internal {
518           require(_isApprovedOrOwner(msg.sender, _tokenId),
519
520           address owner = ownerOf(_tokenId);
521
522           // Clear approval
523           approve(address(0), _tokenId);
524           // TODO add delegates
525           // Remove token
526           _removeTokenFrom(msg.sender, _tokenId);
527           emit Transfer(owner, address(0), _tokenId);
528       }
```

[VotingEscrow.sol#L517-L528]

A user can deposit a token, lock it, wait for the lock to expire, transfer the token to another address, and repeat. During each iteration, a new NFT is minted and checkpointed. Calls to `getPastVotes()` will show the wrong values, since it will think the account still holds the delegation of the burnt NFT. Bribes and gauges also look at the checkpoints and will also have the wrong information

Recommended Mitigation Steps

Call `_moveTokenDelegates(owner,address(0))` in `_burn()`

pooltypes (Velodrome) confirmed and commented:

> Nice catch! We intended to fix this issue (see TODO), included in our mainnet deploy. Thanks for surfacing.

Alex the Entreprenerd (judge) commented:

> The warden has shown an exploit that, leveraging the `_moveTokenDelegates` function, which is not present in `burn` can allow any attacker to inflate their votes.

> The sponsor has confirmed and they indeed have mitigated the issue in their deployed code.

[H-02] `VotingEscrow` 's merge and withdraw aren't available for approved users

Submitted by hyh, also found by hansfriese, rotcivegaf, and WatchPug

Users who are approved, but do not own a particular NFT, are supposed to be eligible to call merge and withdraw from the NFT.

Currently _burn(), used by merge() and withdraw() to remove the NFT from the system, will revert unless the sender is the owner of NFT as the function tries to update the accounting for the sender, not the owner.

Setting the severity to medium as the impact is merge() and withdraw() permanent unavailability for any approved sender, who isn't the owner of the involved NFT.

Proof of Concept

_removeTokenFrom() requires `_from` to be the NFT owner as it removes `_tokenId` from the `_from` account:

[VotingEscrow.sol#L504-L515]

```
    /// @dev Remove a NFT from a given address
    ///      Throws if `_from` is not the current owner.
    function _removeTokenFrom(address _from, uint _tokenId) inte
        // Throws if `_from` is not the current owner
        assert(idToOwner[_tokenId] == _from);
        // Change the owner
        idToOwner[_tokenId] = address(0);
        // Update owner token index tracking
        _removeTokenFromOwnerList(_from, _tokenId);
        // Change count tracking
        ownerToNFTokenCount[_from] -= 1;
    }
```

_burn() allows `_tokenId` to approved or owner, but calls _removeTokenFrom() with `msg.sender` as `_from`:

VotingEscrow.sol#L517-L528

```
    function _burn(uint _tokenId) internal {
        require(_isApprovedOrOwner(msg.sender, _tokenId), "calle

        address owner = ownerOf(_tokenId);

        // Clear approval
        approve(address(0), _tokenId);
        // TODO add delegates
        // Remove token
        _removeTokenFrom(msg.sender, _tokenId);
        emit Transfer(owner, address(0), _tokenId);
    }
```

This way if _burn() is called by an approved account who isn't an owner, it will revert on _removeTokenFrom()'s `assert(idToOwner[_tokenId] == _from)` check.

Now burn() is used by merge():

VotingEscrow.sol#L1084-L1097

```
    function merge(uint _from, uint _to) external {
        require(attachments[_from] == 0 && !voted[_from], "attac
        require(_from != _to);
        require(_isApprovedOrOwner(msg.sender, _from));
        require(_isApprovedOrOwner(msg.sender, _to));

        LockedBalance memory _locked0 = locked[_from];
        LockedBalance memory _locked1 = locked[_to];
        uint value0 = uint(int256(_locked0.amount));
        uint end = _locked0.end >= _locked1.end ? _locked0.end :

        locked[_from] = LockedBalance(0, 0);
        _checkpoint(_from, _locked0, LockedBalance(0, 0));
        _burn(_from);
```

And withdraw():

VotingEscrow.sol#L842-L864

```
    /// @notice Withdraw all tokens for `_tokenId`
    /// @dev Only possible if the lock has expired
    function withdraw(uint _tokenId) external nonreentrant {
        assert(_isApprovedOrOwner(msg.sender, _tokenId));
        require(attachments[_tokenId] == 0 && !voted[_tokenId],

        LockedBalance memory _locked = locked[_tokenId];
        require(block.timestamp >= _locked.end, "The lock didn't
        uint value = uint(int256(_locked.amount));

        locked[_tokenId] = LockedBalance(0,0);
        uint supply_before = supply;
        supply = supply_before - value;

        // old_locked can have either expired <= timestamp or ze
        // _locked has only 0 end
        // Both can have >= 0 amount
        _checkpoint(_tokenId, _locked, LockedBalance(0,0));

        assert(IERC20(token).transfer(msg.sender, value));

        // Burn the NFT
        _burn(_tokenId);
```

Recommended Mitigation Steps

Consider changing _removeTokenFrom() argument to be the owner:

[VotingEscrow.sol#L517-L528]

```
    function _burn(uint _tokenId) internal {
        require(_isApprovedOrOwner(msg.sender, _tokenId), "calle

        address owner = ownerOf(_tokenId);

        // Clear approval
        approve(address(0), _tokenId);
        // TODO add delegates
        // Remove token
-       _removeTokenFrom(msg.sender, _tokenId);
+       _removeTokenFrom(owner, _tokenId);
        emit Transfer(owner, address(0), _tokenId);
```

```
        }
```

pooltypes (Velodrome) disputed

Alex the Entreprenerd (judge) increased severity to High and commented:

> The warden has shown how an approved user is unable to execute ordinary operations due to a logic flaw. While the impact may make Medium Severity valid, as the owner can still operate, but delegated users cannot, I believe the finding shows a logical flaw in the system in that it doesn't work as intended.

> For that reason I believe this finding is of High Severity.

[H-03] User rewards stop accruing after any `_writeCheckpoint` calling action

Submitted by hyh, also found by smiling_heretic, unforgiven, and xiaoming90

Any user balance affecting action, i.e. deposit, withdraw/withdrawToken or getReward, calls _writeCheckpoint to update the balance records used for the earned reward estimation. The issue is that _writeCheckpoint always sets false to `voted` flag for the each new checkpoint due to wrong index used in the mapping access, while only voted periods are eligible for accruing the rewards.

This way any balance changing action of a voted user will lead to stopping of the rewards accrual for the user, until next vote will be cast. I.e. any action that has no relation to voting and should have only balance change as the reward accruing process impact, in fact removes any future rewards from the user until the next vote.

Setting the severity to be high as the impact here violates system logic and means next periods accrued rewards loss for a user.

Proof of Concept

_writeCheckpoint adds a new checkpoint if block.timestamp is not found in the last checkpoint:

Gauge.sol#L302-L313

```
function _writeCheckpoint(address account, uint balance) int
    uint _timestamp = block.timestamp;
    uint _nCheckPoints = numCheckpoints[account];

    if (_nCheckPoints > 0 && checkpoints[account][_nCheckPoi
        checkpoints[account][_nCheckPoints - 1].balanceOf =
    } else {
        bool prevVoteStatus = (_nCheckPoints > 0) ? checkpoi
        checkpoints[account][_nCheckPoints] = Checkpoint(_ti
        numCheckpoints[account] = _nCheckPoints + 1;
    }
}
```

However, instead of moving vote status from the previous checkpoint it records `false` to `prevVoteStatus` all the time as last status is `checkpoints[account][_nCheckPoints-1].voted`, while `checkpoints[account][_nCheckPoints]` isn't created yet and is empty:

Gauge.sol#L309

```
bool prevVoteStatus = (_nCheckPoints > 0) ? checkpoints[
```

Notice that `checkpoints` is a mapping and no range check violation happens:

Gauge.sol#L74-L75

```
/// @notice A record of balance checkpoints for each account
mapping (address => mapping (uint => Checkpoint)) public che
```

This will effectively lead to rewards removal on any user action, as earned() used in rewards estimation counts only voted periods:

Gauge.sol#L483-L502

```
        if (_endIndex > 0) {
            for (uint i = _startIndex; i < _endIndex; i++) {
                Checkpoint memory cp0 = checkpoints[account][i];
                Checkpoint memory cp1 = checkpoints[account][i+1
                (uint _rewardPerTokenStored0,) = getPriorRewardP
                (uint _rewardPerTokenStored1,) = getPriorRewardP
                if (cp0.voted) {
                    reward += cp0.balanceOf * (_rewardPerTokenSt
                }
            }
        }

        Checkpoint memory cp = checkpoints[account][_endIndex];
        uint lastCpWeeksVoteEnd = cp.timestamp - (cp.timestamp %
        if (block.timestamp > lastCpWeeksVoteEnd) {
            (uint _rewardPerTokenStored,) = getPriorRewardPerTok
            if (cp.voted) {
                reward += cp.balanceOf * (rewardPerToken(token)
            }
        }
```

I.e. if a user has voted, then performed any of the operations that call _writeCheckpoint update: deposit, withdraw/withdrawToken or getReward, then this user will not have any rewards for the period between this operation and the next vote as all checkpoints that were created by _writeCheckpoint will have `voted == false`.

Recommended Mitigation Steps

Update the index to be `_nCheckPoints-1`:

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Gauge.sol#L309

```
-       bool prevVoteStatus = (_nCheckPoints > 0) ? checkpoints[
+       bool prevVoteStatus = (_nCheckPoints > 0) ? checkpoints[
```

pooltypes (Velodrome) disputed

Alex the Entreprenerd (judge) commented:

> Due to 1 typo / oversight in the line `bool prevVoteStatus = (_nCheckPoints > 0) ? checkpoints[account][_nCheckPoints].voted : false;` All future checkpoints are registered as having `voted` set to false.

> Due to a configuration choice (or mistake as detailed in other findings), having `voted` set to false causes all rewards to not be receiveable.

> While the impact is loss of Yield (typically a medium finding), the finding shows how this bug will systematically impact all gauges for all users.

> Because of that, I believe High Severity to be appropriate.

[H-04] Bribe Rewards Struck In Contract If Deposited During First Epoch

Submitted by xiaoming90

Bribe rewards added to the `Bribe` contract in the first epoch will not be claimable by any voters, and the rewards will struck in the `Bribe` contract.

Proof-of-Concept

Assume that the current epoch is `epoch 0`, and start date of `epoch 0` is `Day 0`.

When a briber adds a new rewards by calling `Bribe.notifyRewardAmount()`, the `Bribe.getEpochStart()` will return the `start date of current epoch (epoch 0) + 1 day (Bribe Lag)`

Thus, `adjustedTstamp` will be set to `Day 1`. `tokenRewardsPerEpoch[token][adjustedTstamp]` will evaluate to `tokenRewardsPerEpoch[DAI][Day 1]` and the briber's rewards will be stored in `tokenRewardsPerEpoch[DAI][Day 1]`

Bribe.sol#L35

```
function getEpochStart(uint timestamp) public view returns (uint
        uint bribeStart = timestamp - (timestamp % (7 days)) + I
        uint bribeEnd = bribeStart + DURATION - COOLDOWN;
        return timestamp < bribeEnd ? bribeStart : bribeStart +
}
```

[Bribe.sol#L41]

```
function notifyRewardAmount(address token, uint amount) external
  require(amount > 0);
  if (!isReward[token]) {
        require(rewards.length < MAX_REWARD_TOKENS, "too many re
  }
  // bribes kick in at the start of next bribe period
  uint adjustedTstamp = getEpochStart(block.timestamp);
  uint epochRewards = tokenRewardsPerEpoch[token][adjustedTstamp

  _safeTransferFrom(token, msg.sender, address(this), amount);
  tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards + a

  if (!isReward[token]) {
        isReward[token] = true;
        rewards.push(token);
        IGauge(gauge).addBribeRewardToken(token);
  }

  emit NotifyReward(msg.sender, token, adjustedTstamp, amount);
}
```

On `Day 6`, the `voting` phase has ended and the state is currently in the `reward` phase. Alice decided to call the `Voter.distribute` to trigger the distribution of bribe rewards.

However, the main issue is that calling the `Voter.distribute` function on Epoch 0's Day 6 (Reward Phase) will not executed the `Gauge.deliverBribes()` because `claimable[_gauge]` or `_claimable` is currently `0`.

`Gauge.deliverBribes()` is the main function responsible for distributing bribe rewards. Since `Gauge.deliverBribes()` cannot be triggered, the bribe rewards are

forever struck in the `Bribe` Contract.

`claimable[_gauge]` will always be zero on the first epoch because the gauge rewards will only come in the later epoch. The value of `claimable[_gauge]` will only increase when the `Minter.update_period()` function starts minting VELO and distribute them to the gauges. Per the source code of `Minter` contract, the VELO emission will only start from third epoch onwards (`active_period = ((block.timestamp + (2 * WEEK)) / WEEK) * WEEK;`). Thus, before the VELO emission, `claimable[_gauge]` will always remain at `0` .

Voter.sol#L315

```
    function distribute(address _gauge) public lock {
            require(isAlive[_gauge]); // killed gauges cannot distri
            uint dayCalc = block.timestamp % (7 days);
            require((dayCalc < BRIBE_LAG) || (dayCalc > (DURATION +
            IMinter(minter).update_period();
            _updateFor(_gauge);
            uint _claimable = claimable[_gauge];
            if (_claimable > IGauge(_gauge).left(base) && _claimable
                    claimable[_gauge] = 0;
                    IGauge(_gauge).notifyRewardAmount(base, _claimab
                    emit DistributeReward(msg.sender, _gauge, _claim
                    // distribute bribes & fees too
                    IGauge(_gauge).deliverBribes();
            }
    }
```

If someone attempt to call `Voter.distribute()` on epoch 1 or subsequent epoch, it will fetch the bribe rewards in their respective epoch.

In the `Gauge.deliverBribes` function, the code `uint bribeStart = block.timestamp - (block.timestamp % (7 days)) + BRIBE_LAG;` will calculate the start date of current epoch + BRIBE_LAG (1 day). So, if someone call `Gauge.deliverBribes` in epoch 1, the `bribeStart` variable will be set to the `Epoch 1 + 1 day` , which is equivalent to Day 9. There is no way to fetch the bribe rewards struck in epoch 0.

Gauge.sol#L173

```
    function deliverBribes() external lock {
            require(msg.sender == voter);
            IBribe sb = IBribe(bribe);
            uint bribeStart = block.timestamp - (block.timestamp %
            uint numRewards = sb.rewardsListLength();

            for (uint i = 0; i < numRewards; i++) {
                    address token = sb.rewards(i);
                    uint epochRewards = sb.deliverReward(token, brib
                    if (epochRewards > 0) {
                            _notifyBribeAmount(token, epochRewards,
                    }
            }
    }
```

Bribe.sol#L83

```
    function deliverReward(address token, uint epochStart) external
            require(msg.sender == gauge);
            uint rewardPerEpoch = tokenRewardsPerEpoch[token][epochS
            if (rewardPerEpoch > 0) {
              _safeTransfer(token, address(gauge), rewardPerEpoch);
            }
            return rewardPerEpoch;
    }
```

Recommended Mitigation Steps

Implement logic to handle the edge case where bribe rewards are added during first epoch. Consider aligning the start of bribe period with VELO emission period.

pooltypes (Velodrome) acknowledged

Alex the Entreprenerd (judge) commented:

> The warden has shown how tokens can be stuck in the Bribes Contract indefinitely.

> This is because bribes can be added at the beginning (day 0), while rewards can be received only after a delay (7 days), due to the need for a minimum amount of rewards to be available in order for bribes to be claimable, the logic will prevent the bribes deposited on day 0 to be claimable.

> I'm conflicted on the severity as technically this can only happen for the first week, however the loss of tokens is irreversible as there's no code that would allow rescuing them.

> On further consideration, because the deployment of new Bribes and Gauges is mostly permissioneless, this bug will be present for every new pair of deployed contract, and it is highly likely that a new protocol would want to add rewards immediately.

> For those reasons, I believe High Severity to be appropriate.

[H-05] Voting overwrites `checkpoint.voted` in last checkpoint, so users can just vote right before claiming rewards

Submitted by smiling_heretic

Gauge.sol#L195
Gauge.sol#L489-L490
Gauge.sol#L499-L500

```
if (cp0.voted) {
    reward += cp0.balanceOf * (_rewardPerTokenStorec
```

this line in `gauge.earned` function looks like the intention here is to incentivize users to keep their `escrow.balanceOfNft` voted for this gauge.

However, it's enough to vote just before claiming rewards (even in the same transaction) and `voter.reset` just after receiving rewards to pass this `if` and get rewards for full period since last interaction with the gauge.

Proof of Concept

See original submission for test file.

Note, that Bob kept his votes for this gauge for full 6-day period but Alice just voted before claiming rewards. In logs, we can see that they both received the same (non-zero) amount of VELO tokens.

Alice can reset her votes in the same transaction after claiming rewards, if she decides to do so.

Tools Used

Foundry

Recommended Mitigation Steps

A partial solution would be to create a new checkpoint each time user's `voted` status changes (`setVoteStatus` is called) instead of overwriting the `voted` in last one.

However, even then, users can just assign very small weight to this gauge, and lock very little VELO, so I don't think this `if` statement helps with anything. I think, it's better to rethink how to incentivize users to vote for specific gauges.

pooltypes (Velodrome) acknowledged and commented:

> Patched in mainnet deployment.

Alex the Entreprenerd (judge) increased severity to High and commented:

> The warden has found a way to sidestep the loss of rewards that automatically happens due to the faulty checkpoint system that always sets voted to false.

> In doing so they also showed how the system can fall apart and provided a POC to replicate.

> Because I've rated issues related to the `voted` checkpoints and loss of rewards with High Severity, at this time I believe this finding should also be bumped as it shows how the system is broken and the way to avoid a loss of rewards.

> The sponsor seems to have remedied by deleting the voted logic.

[H-06] Attacker can block LayerZero channel

Submitted by Ruhum

According to the LayerZero docs, the default behavior is that when a transaction on the destination application fails, the channel between the src and dst app is blocked. Before any new transactions can be executed, the failed transaction has to be retried until it succeeds.

See https://layerzero.gitbook.io/docs/faq/messaging-properties#message-ordering & https://layerzero.gitbook.io/docs/guides/advanced/nonblockinglzapp

So an attacker is able to initiate a transaction they know will fail to block the channel between FTM and Optimism. The RedemptionSender & Receiver won't be usable anymore.

Proof of Concept

The RedemptionReceiver contract doesn't implement the non-blocking approach as seen here:
https://github.com/code-423n4/2022-05-velodrome/blob/main/contracts/contracts/redeem/RedemptionReceiver.sol#L72-L105

An example implementation of the non-blocking approach by LayerZero:
https://github.com/LayerZero-Labs/solidity-examples/blob/main/contracts/lzApp/NonblockingLzApp.sol

Recommended Mitigation Steps

Use the non-blocking approach as described here.

pooltypes (Velodrome) disagreed with severity

Alex the Entreprenerd (judge) commented:

> @pooltypes Can anyone send a message or would they need to be whitelisted?

Alex the Entreprenerd (judge) commented:

> If anyone can call and deny, the contract is not suited to handle exceptions and doesn't implement the `forceReceive` function, meaning the channel can be griefed and I don't believe there's a way to remedy.

> The contract needs to implement `forceResumeReceive` to allow to remove malicious messages that may be received.

> I still am unsure if anyone can send a malicious message or if they need to be approved. If only the admin can this is a Medium Severity. If anyone can, this is a High Severity finding.

> From the documentation it seems like anyone can call the function: https://layerzero.gitbook.io/docs/guides/master/how-to-send-a-message

Alex the Entreprenerd (judge) increased severity to High and commented:

> With the information I currently have, it seems like the channel can be setup to receive messages only by the specified contract, however for multiple reasons, the message sent can cause a revert, and in lack of a "nonblocking" architecture, the messages can get stuck indefinitely.

> However, the implementation under scope has none of these defenses, it seems like the contact under scope can be denied functionality by any caller that builds their own LZApp.

> See example of how to prevent untrusted callers.

> Because of that, the message queue can be filled with blocking messages that cannot be removed.

> Because the contract under scope also has no way of re-setting the queue, I have reason to believe that any attack can permanently brick the receiver.

> For these reasons, I believe High Severity to be more appropriate.

ethzoomer (Velodrome) commented:

> At this point in time we've already completed all of the redemptions.

> Is it possible to send a message from the contract other than what sender sends? lz's msg queues are per src addr. https://layerzero.gitbook.io/docs/faq/messaging-properties "STORED message will block the delivery of any future message from srcUA to all dstUA on the same destination chain and can be retried until the message becomes SUCCESS" The only way that can get gummed up is if redemption's over, right?

Alex the Entreprenerd (judge) commented:

> My understanding is any sender can block the queue as the receiver will revert.

> That said if redemption is over, there's no loss beside the risk of burning funds from the FTM side.

Medium Risk Findings (17)

[M-01] Gauge set can be front run if bribe and gauge constructors aren't run atomically

Submitted by hyh, also found by 0x1f8b

If Bribe and Gauge constructors are run not in the same transaction, the griefing attack is possible. A malicious user can run setGauge after Bribe, but before Gauge constructor, making Bribe contract unusable. The fix here is Bribe redeployment.

Setting severity to be medium as that is temporary system breaking impact.

Proof of Concept

setGauge can be run by anyone, but only once with a meaningful gauge:

Bribe.sol#L30-L33

```
function setGauge(address _gauge) external {
  require(gauge == address(0), "gauge already set");
```

```
        gauge = _gauge;
    }
```

Now it is called in Gauge constructor:

[Gauge.sol#L96-L104]

```
    constructor(address _stake, address _bribe, address  __ve, a
        stake = _stake;
        bribe = _bribe;
        _ve = __ve;
        voter = _voter;

        factory = msg.sender;

        IBribe(bribe).setGauge(address(this));
```

This way it will not be called before Gauge constructor, but if it is not atomic with Bribe constructor, an attacker can call in-between.

Recommended Mitigation Steps

Consider either running Bribe and then Gauge constructors atomically, or introducing an owner role in Bribe constructor and onlyOwner access control in setGauge, setting it manually.

pooltypes (Velodrome) confirmed and commented:

> Thanks, we've addressed this and removed the `setGauge` pattern in our mainnet deployment.

Alex the Entreprenerd (judge) commented:

> The warden has identified a grief that can be done by front-running a non-authorized call to `setGauge`.

> Impact would require re-deploying the bribes contract and in lack of an atomic deploy + set. I believe the finding to be valid and Medium Severity to be

> appropriate.

[M-02] `VeloGovernor: proposalNumerator` and team are updated by team, not governance

Submitted by cccz

setProposalNumerator and setTeam functions of the VeloGovernor contract can only be called by team without allowing governance to call.

Proof of Concept

[VeloGovernor.sol#L44-L48]

Recommended Mitigation Steps

Allow governance to call setProposalNumerator and setTeam functions.

pooltypes (Velodrome) acknowledged

Alex the Entreprenerd (judge) commented:

> I really dislike how the finding is phrased as the variable `team` doesn't mean much by itself.

> That said the finding has validity in that the Governance Contract should be calling to change the `proposalNumerator` so a check should probably be for `address(this)`.

[M-03] Alter velo receptions computation

Submitted by 0x1f8b, also found by hansfriese

The `deployer` can modify the values to obtain different values than expected.

Proof of Concept

The flag used in `initializeReceiverWith` to prevent re-initialization is `fantomSender == address(0)`, however the provided address for that value is not checked to be different from `address(0)`, so it could be initialized multiple times to establish that value.

This could allow minting multiple Velos into Velo.mintToRedemptionReceiver and setting the wrong value in `redeemableVELO` and `redeemableUSDC` to affect the calculations of the `previewRedeem` method and later receive larger amounts of tokens in `lzReceive` than expected.

Affected source code:

- RedemptionReceiver.sol#L44

Recommended Mitigation Steps

- Ensure that the provided addresses are not `address(0)`.

pooltypes (Velodrome) acknowledged

Alex the Entreprenerd (judge) commented:

> The warden has shown how, through a combination of misconfiguration and admin privilege, more tokens than expected could be transfered.

> In looking at `Velo` and `Minter` we can see that `RedemtionReceiver` has to be set by `minter` which at deployment time will be the deployer.

> I don't believe the exploit can be applied after a correct configuration as `Minter` is unable to set a new `RedemptionReceiver` meaning only the original deployer could pull of the exploit only until they set `minter` to `Minter.sol`.

> Because of the external requirements, Medium Severity is appropriate; however, I believe the likelihood of this attack being pulled off is minimal.

[M-04] Malicious user can populate `rewards` array with

tokens of their interest reaching limits of `MAX_REWARD_TOKENS`

Submitted by 0xf15ers, also found by 0x52, berndartmueller, cccz, horsefacts, hyh, minhquanym, pauliax, Ruhum, and WatchPug

Malicious user can populate `rewards` array with different tokens early reaching limit of `MAX_REWARD_TOKENS` sending very small amount of different tokens. It will restrict any other tokens to be used as `rewards` in Bribe.sol#notifyRewardAmount()

Proof of Concept

A custom malicious contract can be created that can make multiple calls to `notifyRewardAmount()` sending very small amounts of different tokens to populate the array `rewards` and fulfill the total of `MAX_REWARD_TOKENS` . This will restrict any other person from adding to `rewards` array.

pooltypes (Velodrome) confirmed and commented:

> Fixed. We added a whitelist check for reward tokens for our mainnet deployment.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> Given that the deployer can re-deploy contracts at any time, and they also can multi-call, I don't believe a realistic DOS can happen.
>
> I do, however, agree that any additional reward beside the first token (which can be automatically added via a deploy + initialize + distribute), may end up being denied.
>
> Additionally the `team` can use `swapOutBribeRewardToken` to change to a real reward token.
>
> Because of this, I believe the finding to be valid and of Medium Severity.

[M-05] `Bribe.sol` is not meant to handle fee-on-transfer tokens

Submitted by MiloTruck, also found by 0x52, Dravee, IIIIIII, MaratCerby, unforgiven, and WatchPug

[Bribe.sol#L50-L51](#)
[Bribe.sol#L83-L90](#)

Should a fee-on-transfer token be added as a reward token and deposited, the tokens will be locked in the `Bribe` contract. Voters will be unable to withdraw their rewards.

Proof of Concept

Tokens are deposited into the `Bribe` contract using `notifyRewardAmount()`, where `amount` of tokens are transferred, then added directly to `tokenRewardsPerEpoch[token][adjustedTstamp]`:

```
        _safeTransferFrom(token, msg.sender, address(this), amount);
        tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards +
```

Tokens are transferred out of the `Bribe` contract using `deliverReward()`, which attempts to transfer `tokenRewardsPerEpoch[token][epochStart]` amount of tokens out.

```
    function deliverReward(address token, uint epochStart) external
        require(msg.sender == gauge);
        uint rewardPerEpoch = tokenRewardsPerEpoch[token][epochStart
        if (rewardPerEpoch > 0) {
            _safeTransfer(token, address(gauge), rewardPerEpoch);
        }
        return rewardPerEpoch;
    }
```

If `token` happens to be a fee-on-transfer token, `deliverReward()` will always fail. For example:

- User calls `notifyRewardAmount()`, with `token` as token that charges a 2% fee upon any transfer, and `amount = 100`:

    - `_safeTransferFrom()` only transfers 98 tokens to the contract due to the 2% fee
    - Assuming `epochRewards = 0`, `tokenRewardsPerEpoch[token][adjustedTstamp]` becomes `100`
- Later on, when `deliverReward()` is called with the same `token` and `epochStart`:
    - `rewardPerEpoch = tokenRewardsPerEpoch[token][epochStart] = 100`
    - `_safeTransfer` attempts to transfer 100 tokens out of the contract
    - However, the contract only contains 98 tokens
    - `deliverReward()` reverts

The following test, which implements a [MockERC20 with fee-on-transfer], demonstrates this:

```
// Note that the following test was adapted from Bribes.t.sol
function testFailFeeOnTransferToken() public {
    // Deploy ERC20 token with fee-on-transfer
    MockERC20Fee FEE_TOKEN = new MockERC20Fee("FEE", "FEE", 18);

    // Mint FEE token for address(this)
    FEE_TOKEN.mint(address(this), 1e25);

    // vote
    VELO.approve(address(escrow), TOKEN_1);
    escrow.create_lock(TOKEN_1, 4 * 365 * 86400);
    vm.warp(block.timestamp + 1);

    address[] memory pools = new address[](1);
    pools[0] = address(pair);
    uint256[] memory weights = new uint256[](1);
    weights[0] = 10000;
    voter.vote(1, pools, weights);

    // and deposit into the gauge!
    pair.approve(address(gauge), 1e9);
    gauge.deposit(1e9, 1);

    vm.warp(block.timestamp + 12 hours); // still prior to epoch
    vm.roll(block.number + 1);
```

```
        assertEq(uint(gauge.getVotingStage(block.timestamp)), uint(0

        vm.warp(block.timestamp + 12 hours); // start of epoch
        vm.roll(block.number + 1);
        assertEq(uint(gauge.getVotingStage(block.timestamp)), uint(0

        vm.warp(block.timestamp + 5 days); // votes period over
        vm.roll(block.number + 1);

        vm.warp(2 weeks + 1); // emissions start
        vm.roll(block.number + 1);

        minter.update_period();
        distributor.claim(1); // yay this works

        vm.warp(block.timestamp + 1 days); // next votes period star
        vm.roll(block.number + 1);

        // get a bribe
        owner.approve(address(FEE_TOKEN), address(bribe), TOKEN_1);
        bribe.notifyRewardAmount(address(FEE_TOKEN), TOKEN_1);

        vm.warp(block.timestamp + 5 days); // votes period over
        vm.roll(block.number + 1);

        // Atttempt to claim tokens will revert
        voter.distro(); // bribe gets deposited in the gauge
    }
```

Additional Impact

On a larger scale, a malicious attacker could temporarily DOS any `Gauge` contract. This can be done by:

1. Depositing a fee-on-transfer token into its respective `Bribe` contract, using `notifyRewardAmount()` , and adding it as a reward token.
2. This would cause `deliverBribes()` to fail whenever it is called, thus no one would be able to withdraw any reward tokens from the `Gauge` contract.

The only way to undo the DOS would be to call `swapOutBribeRewardToken()` and swap out the fee-on-transfer token for another valid token.

Recommended Mitigation

- The amount of tokens received should be added to `epochRewards` and stored in `tokenRewardsPerEpoch[token][adjustedTstamp]`, instead of the amount stated for transfer. For example:

```
uint256 _before = IERC20(token).balanceOf(address(this));
_safeTransferFrom(token, msg.sender, address(this), amount);
uint256 _after = IERC20(token).balanceOf(address(this));

tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards +
```

- Alternatively, disallow tokens with fee-on-transfer mechanics to be added as reward tokens.

pooltypes (Velodrome) acknowledged and commented:

> Reward tokens are now whitelisted in our mainnet deployment.

Alex the Entreprenerd (judge) commented:

> The warden has shown how a feeOnTransfer token can cause accounting issues and cause a loss of rewards for end users.

> Because of the open-ended nature of the bribes contract, as well as the real risk of loss of promised rewards, I believe the finding to be valid and of Medium Severity.

[M-06] Voting tokens may be lost when given to non-EOA accounts

Submitted by lllllll, also found by rotcivegaf and Ruhum

veNFTs may be sent to contracts that cannot handle them, and therefore all rewards and voting power, as well as the underlying are locked forever

Proof of Concept

The original code had the following warning:

```
     * @dev Safely transfers `tokenId` token from `from` to `to`,
     * are aware of the ERC721 protocol to prevent tokens from be
```

[ve.sol#L143-L144]

The minting code, which creates the locks, does not do this check:

File: contracts/contracts/VotingEscrow.sol #1

```
462       function _mint(address _to, uint _tokenId) internal re
463           // Throws if `_to` is zero address
464           assert(_to != address(0));
465           // TODO add delegates
466           // checkpoint for gov
467           _moveTokenDelegates(address(0), delegates(_to), _t
468           // Add NFT. Throws if `_tokenId` is owned by somec
469           _addTokenTo(_to, _tokenId);
470           emit Transfer(address(0), _to, _tokenId);
471           return true;
472       }
```

[VotingEscrow.sol#L462-L472]

Once a lock is already minted, if it's transfered to a contract, the code does attempt to check for the issue:

File: contracts/contracts/VotingEscrow.sol #2

```
378       ///      If `_to` is a smart contract, it calls `onERC
379       ///      the return value is not `bytes4(keccak256("or
380       /// @param _from The current owner of the NFT.
381       /// @param _to The new owner.
382       /// @param _tokenId The NFT to transfer.
383       /// @param _data Additional data with no specified for
384       function safeTransferFrom(
385           address _from,
386           address _to,
387           uint _tokenId,
```

```
388             bytes memory _data
389         ) public {
390             _transferFrom(_from, _to, _tokenId, msg.sender);
391 
392             if (_isContract(_to)) {
393                 // Throws if transfer destination is a contrac
394                 try IERC721Receiver(_to).onERC721Received(msg.
395                     bytes memory reason
396                 ) {
397                     if (reason.length == 0) {
398                         revert('ERC721: transfer to non ERC721
399                     } else {
400                         assembly {
401                             revert(add(32, reason), mload(reas
402                         }
403                     }
404                 }
405             }
406         }
```

[VotingEscrow.sol#L378-L406]

While the transfer function does in fact execute `onERC721Received`, it doesn't actually do a check of the `bytes4` variable - it only checks for a non-zero length. The ERC721 standard says that for the function call to be valid, it must return the `bytes4` function selector, otherwise it's invalid. If a user of the escrow system uses a contract that is attempting to explicitly reject NFTs by returning zero in its `onERC721Received()` function, the `VotingEscrow` will interpret that response as success and will transfer the NFT, potentially locking it forever. If the lock is minted to a contract, no checks are done, and the NFT can be locked forever.

Recommended Mitigation Steps

Call `onERC721Received()` in `_mint()` and ensure that the return value equals `IERC721Receiver.onERC721Received.selector` in both `_mint()` and `safeTransferFrom()`.

pooltypes (Velodrome) disputed

Alex the Entreprenerd (judge) commented:

> While I would be surprised by any realistic scenario of a smart contract that would implement `onERC721Received` and not return the specified selector, I have to acknowledge that technically we can get multiple false positives, especially when dealing with contracts that have a `fallback` function .

> That said, to me that was not sufficient to constitute a valid finding.

> However, I went and checked the EIP-721 and the standard definition of `safeTransferFrom` , which can be verified here: https://eips.ethereum.org/EIPS/eip-721

```
/// @notice Transfers the ownership of an NFT from one address to another address
/// @dev Throws unless `msg.sender` is the current owner, an authorized
///  operator, or the approved address for this NFT. Throws if `_from` is
///  not the current owner. Throws if `_to` is the zero address. Throws if
///  `_tokenId` is not a valid NFT. When transfer is complete, this function
///  checks if `_to` is a smart contract (code size > 0). If so, it calls
///  `onERC721Received` on `_to` and throws if the return value is not
///  `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`.
/// @param _from The current owner of the NFT
/// @param _to The new owner
/// @param _tokenId The NFT to transfer
/// @param data Additional data with no specified format, sent in call to `_to`
function safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data) external payable;
```

> According to the standard the function must check for the return value, and for that reason I believe the finding to be valid.

> While I would be surprised if this causes any issue in the real world, because the function is non-conformant to the standard and technically a loss can happen because of it, I believe Medium Severity to be appropriate.

> To confirm, I doubt any meaningful loss will ever happen.

> However this finding is basically: “Incorrect implementation of `onERC721Received` ” and it’s a valid finding at that.

[M-07] `RedemptionSender` should estimate fees to prevent failed transactions

Submitted by Ruhum

RedemptionSender.sol#L43

When sending a msg to the layer zero endpoint you include enough gas for the transaction. If you don't include enough tokens for the gas, the transaction will fail. The RedemptionSender contract allows the user to pass any value they want which might result in them sending not enough. Their transaction will fail.

To know how much you have to send there's the `estimateFees()` function as described here.

Proof of Concept

```
function redeemWEVE(
    uint256 amount,
    address zroPaymentAddress,
    bytes memory zroTransactionParams
) public payable {
    require(amount != 0, "AMOUNT_ZERO");
    require(
        IERC20(weve).transferFrom(
            msg.sender,
            0x000000000000000000000000000000000000dEaD,
            amount
        ),
        "WEVE: TRANSFER_FAILED"
    );

    ILayerZeroEndpoint(endpoint).send{value: msg.value}(
        optimismChainId,
        abi.encodePacked(optimismReceiver),
        abi.encode(msg.sender, amount),
        payable(msg.sender),
        zroPaymentAddress,
        zroTransactionParams
    );
}
```

Recommended Mitigation Steps

Use the `estimateFees()` endpoint.

pooltypes (Velodrome) acknowledged

Alex the Entreprenerd (judge) commented:

> The transaction shown has a case in which it could succeed but the message wouldn't be forwarded.

> That is because any estimate_gas will make the tx work but the gas payment may not be sufficient to fund the message fees.

> Because this can cause burned tokens, I believe a check should be added and since the tx must originate from the `fantomSender` there would be no way of forwarding the message on the behalf of the sender.

> For those reasons I believe the finding is of Medium Severity.

[M-08] Temporary DOS by calling `notifyRewardAmount()` in Bribe/Gauge with malicious tokens

Submitted by unforgiven, also found by 0x52 and Picodes

Bribe.sol#L41-L60
Gauge.sol#L590-L624

It's possible to call `notifyRewardAmount()` in `Bribe` or `Gauge` contract with malicious tokens and contract will add them to reward tokens list and then attacker can interrupt `Gauge.deliverBribes()` logic (by failing all contract transaction in that malicious token). because `Gauge.deliverBribes()` is looping through all tokens and transferring rewards and if one of them fails whole transaction will fail. As `Gauge.deliverBribes()` is called by `Voter.distribute()` and `Voter.distribute()` is called by `Gauge.getReward()` so functions: `Voter.distribute()` and `Gauge.getReward()` will be broke too and no one can call them for that `Gauge`.

Proof of Concept

This is `notifyRewardAmount()` code in `Bribe`:

```
function notifyRewardAmount(address token, uint amount) exterr
    require(amount > 0);
    if (!isReward[token]) {
      require(rewards.length < MAX_REWARD_TOKENS, "too many re
    }
    // bribes kick in at the start of next bribe period
    uint adjustedTstamp = getEpochStart(block.timestamp);
    uint epochRewards = tokenRewardsPerEpoch[token][adjustedTs

    _safeTransferFrom(token, msg.sender, address(this), amount
    tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards

    if (!isReward[token]) {
        isReward[token] = true;
        rewards.push(token);
        IGauge(gauge).addBribeRewardToken(token);
    }

    emit NotifyReward(msg.sender, token, adjustedTstamp, amour
}
```

As you can see it's callable by anyone and it will add the new tokens to `rewards` list in `Bribe` and `Gauge` contract. `notifyRewardAmount()` in `Gauge` is similar to `Bribe`'s `notifyRewardAmount()`.
This is `deliverBribes()` code in `Gauge`:

```
function deliverBribes() external lock {
    require(msg.sender == voter);
    IBribe sb = IBribe(bribe);
    uint bribeStart = block.timestamp - (block.timestamp % (
    uint numRewards = sb.rewardsListLength();

    for (uint i = 0; i < numRewards; i++) {
        address token = sb.rewards(i);
        uint epochRewards = sb.deliverReward(token, bribeSta
        if (epochRewards > 0) {
            _notifyBribeAmount(token, epochRewards, bribeSta
        }
    }
}
```

As you can see it loops through all `rewards` token list and calls `Bribe.deliverReward()` which then transfers the reward token. but if one of the tokens were malicious and revert the transaction then the whole transaction will fail and `deliverBribes()` logic will be blocked. This is `distribute()` code in `Voter`:

```
    function distribute(address _gauge) public lock {
        require(isAlive[_gauge]); // killed gauges cannot distri
        uint dayCalc = block.timestamp % (7 days);
        require((dayCalc < BRIBE_LAG) || (dayCalc > (DURATION +
        IMinter(minter).update_period();
        _updateFor(_gauge);
        uint _claimable = claimable[_gauge];
        if (_claimable > IGauge(_gauge).left(base) && _claimable
            claimable[_gauge] = 0;
            IGauge(_gauge).notifyRewardAmount(base, _claimable);
            emit DistributeReward(msg.sender, _gauge, _claimable
            // distribute bribes & fees too
            IGauge(_gauge).deliverBribes();
        }
    }
```

As you can see it calls `Gauge.deliverBribes()`. and this is `getReward()` code in `Gauge`:

```
    function getReward(address account, address[] memory tokens)
        require(msg.sender == account || msg.sender == voter);
        _unlocked = 1;
        IVoter(voter).distribute(address(this));
        _unlocked = 2;

        for (uint i = 0; i < tokens.length; i++) {
            (rewardPerTokenStored[tokens[i]], lastUpdateTime[tok

            uint _reward = earned(tokens[i], account);
            lastEarn[tokens[i]][account] = block.timestamp;
            userRewardPerTokenStored[tokens[i]][account] = rewar
            if (_reward > 0) _safeTransfer(tokens[i], account, _

            emit ClaimRewards(msg.sender, tokens[i], _reward);
        }
```

```
            uint _derivedBalance = derivedBalances[account];
            derivedSupply -= _derivedBalance;
            _derivedBalance = derivedBalance(account);
            derivedBalances[account] = _derivedBalance;
            derivedSupply += _derivedBalance;

            _writeCheckpoint(account, derivedBalances[account]);
            _writeSupplyCheckpoint();
        }
```

which calls `Voter.distribute()`. so if `Gauge.deliverBribes()` fails then the logic of `Voter.distribute()` and `Gauge.getReward()` will fail too and users couldn't call them and attacker can cause DOS. Of course the `team` can swap those malicious reward tokens added by attacker with `swapOutRewardToken()` and `swapOutBribeRewardToken()` but for some time those logics will not work and attacker can cause DOS.

Tools Used

VIM

Recommended Mitigation Steps

Set access levels for `notifyRewardAmount()` functions.

pooltypes (Velodrome) acknowledged

Alex the Entreprenerd (judge) commented:

> The warden has shown how, through adding a malicious bribe token via `notifyRewardAmount`, it is possible to deny claiming of all bribes.

> An alternative would be to allow claiming directly in the Bribe.sol contract (which I believe is the original Solidly design).

> However, at this time the Sponsor has remediated through a list of approved tokens.

> Because the DOS is contingent on external circumstances and ultimately can be ended by swapping out the bribe token (and filling in spam innocuous bribes), I believe Medium Severity to be appropriate.

[M-09] Owner's delegates should be decreased in `_burn()`

Submitted by WatchPug

[VotingEscrow.sol#L517-L528]

```
    function _burn(uint _tokenId) internal {
        require(_isApprovedOrOwner(msg.sender, _tokenId), "caller is

        address owner = ownerOf(_tokenId);

        // Clear approval
        approve(address(0), _tokenId);
        // TODO add delegates
        // Remove token
        _removeTokenFrom(msg.sender, _tokenId);
        emit Transfer(owner, address(0), _tokenId);
    }
```

[VotingEscrow.sol#L1244-L1248]

```
    // All the same plus _tokenId
    require(
        dstRepOld.length + 1 <= MAX_DELEGATES,
        "dstRep would have too many tokenIds"
    );
```

When `_moveTokenDelegates()`, `dstRep` must not have more than `MAX_DELEGATES`.

However, in `_burn()`, `dstRep` array wont get decreased. This opens a griefing attack vector, in which the attacker can deposit 1 wei of token for the victim, repeated for `MAX_DELEGATES` times, and the victim can no longer deposit, even if they try to merge or burn their tokenIds.

Even for a normal active user, this can be a problem as the `dstRep` array will continuously grow and cant decrease, one day the user won't be able to deposit anymore as they have reached the `MAX_DELEGATES` cap.

Recommendation

Change to:

```
    function _burn(uint _tokenId) internal {
        require(_isApprovedOrOwner(msg.sender, _tokenId), "caller is

        address owner = ownerOf(_tokenId);

        // Clear approval
        approve(address(0), _tokenId);
        // remove delegates
        _moveTokenDelegates(delegates(_to), address(0), _tokenId);
        // Remove token
        _removeTokenFrom(msg.sender, _tokenId);
        emit Transfer(owner, address(0), _tokenId);
    }
```

pooltypes (Velodrome) acknowledged

Alex the Entreprenerd (judge) commented:

> The warden has shown how, in lack of `_moveTokenDelegates(delegates(_to), address(0), _tokenId);` an attacker could deny the ability to deposit tokens in the protocol.

> Because a user could always roll a new address, I agree with Medium Severity.

[M-10] Rewards aren't updated before user's balance change in Gauge's `withdrawToken`

Submitted by hyh, also found by codexploder

_updateRewardForAllTokens is called in withdraw() only, while both withdraw() and withdrawToken() are public and can be called directly. Reward update is needed on user's balance change, its absence leads to reward data rewriting and rewards losing impact for a user.

Per discussion with project withdrawToken() will not be exposed in UI, so setting the severity to be medium as user's direct usage of the vulnerable function is required here, while the impact is losing of a part of accumulated rewards.

Proof of Concept

withdraw() is a wrapper for withdrawToken(), which changes user's balance. withdrawToken() can be called directly, and in this case there will be no rewards update:

Gauge.sol#L548-L561

```
    function withdraw(uint amount) public {
        _updateRewardForAllTokens();

        uint tokenId = 0;
        if (amount == balanceOf[msg.sender]) {
            tokenId = tokenIds[msg.sender];
        }
        withdrawToken(amount, tokenId);
    }

    function withdrawToken(uint amount, uint tokenId) public loc
        totalSupply -= amount;
        balanceOf[msg.sender] -= amount;
        _safeTransfer(stake, msg.sender, amount);
```

This way if a user call withdrawToken(), the associated rewards can be lost due to incomplete reward balance accounting.

References

A showcase of reward update issue: https://github.com/belbix/solidly/issues/1

It's replicated live with users losing accumulated rewards:
https://github.com/solidlyexchange/solidly/issues/55

Recommended Mitigation Steps

Consider moving rewards update to withdrawToken(), since it is called by withdraw() anyway:

```
    function withdraw(uint amount) public {
-       _updateRewardForAllTokens();

        uint tokenId = 0;
        if (amount == balanceOf[msg.sender]) {
            tokenId = tokenIds[msg.sender];
        }
        withdrawToken(amount, tokenId);
    }

    function withdrawToken(uint amount, uint tokenId) public loc
+       _updateRewardForAllTokens();

        totalSupply -= amount;
        balanceOf[msg.sender] -= amount;
        _safeTransfer(stake, msg.sender, amount);
```

Or, if the intent is to keep withdrawToken() as is, consider making it private, so no direct usage be possible:

```
-    function withdrawToken(uint amount, uint tokenId) public lo
+    function withdrawToken(uint amount, uint tokenId) private ]

        totalSupply -= amount;
        balanceOf[msg.sender] -= amount;
        _safeTransfer(stake, msg.sender, amount);
```

pooltypes (Velodrome) disputed

Alex the Entreprenerd (judge) commented:

> I'm extremely conflicted about this finding as it tries to imply that not calling `_updateRewardForAllTokens` is dangerous, yet no effort is made in showing why.

> That said, `_updateRewardForAllTokens` seems to be reliant on the `supplyCheckpoints` which would be changed via `_writeSupplyCheckpoint`. Leading me to agree that the reward math will be wrong.

[M-11] Griefing Attack By Extending The Reward Duration

Submitted by xiaoming90

The `Gauge.notifyRewardAmount` notifies the contract of a newly received rewards. This updates the local accounting and streams the reward over a preset period (Five days).

It was observed that this function is callable by anyone regardless of whether the previous reward period has already expired or not. Thus, it would be possible to exploit the system by repeatedly calling it with dust reward amount to extend an active reward period, and thus dragging out the duration over which the rewards are released.

Since Velodrome is to be deployed on Layer 2 blockchain where the gas fee tends to be cheap, the effort required to carry out this attack would be minimal.

This issue was also highlighted in Curve documentation under the `ChildChainStreamer.notify_reward_amount(token: address):` section.

Gauge.sol#L590

```
    function notifyRewardAmount(address token, uint amount) external
        require(token != stake);
        require(amount > 0);
        if (!isReward[token]) {
            require(rewards.length < MAX_REWARD_TOKENS, "too many re
        }
        // rewards accrue only during the bribe period
        uint bribeStart = block.timestamp - (block.timestamp % (7 da
```

```
        uint adjustedTstamp = block.timestamp < bribeStart ? bribeSt
        if (rewardRate[token] == 0) _writeRewardPerTokenCheckpoint(t
        (rewardPerTokenStored[token], lastUpdateTime[token]) = _upda
        _claimFees();

        if (block.timestamp >= periodFinish[token]) {
            _safeTransferFrom(token, msg.sender, address(this), amou
            rewardRate[token] = amount / DURATION;
        } else {
            uint _remaining = periodFinish[token] - block.timestamp;
            uint _left = _remaining * rewardRate[token];
            require(amount > _left);
            _safeTransferFrom(token, msg.sender, address(this), amou
            rewardRate[token] = (amount + _left) / DURATION;
        }
        require(rewardRate[token] > 0);
        uint balance = IERC20(token).balanceOf(address(this));
        require(rewardRate[token] <= balance / DURATION, "Provided r
        periodFinish[token] = adjustedTstamp + DURATION;
        if (!isReward[token]) {
            isReward[token] = true;
            rewards.push(token);
            IBribe(bribe).addRewardToken(token);
        }

        emit NotifyReward(msg.sender, token, amount);
    }
```

Recommended Mitigation Steps

Consider implementing validation to ensure that this function is callable by anyone only if the previous reward period has already expired. Otherwise, when there is an active reward period, it may only be called by the designated reward distributor account.

For sample implementation, see here.

pooltypes (Velodrome) acknowledged

Alex the Entreprenerd (judge) commented:

> The warden has shown how a reward period could be extended, causing a dilution of reward rate. Because this impact can cause a loss of Yield, I believe Medium

> Severity to be appropriate.

[M-12] Rewards can be locked in Bribe contract because distributing them depends on base token reward amount and `Gauge.deliverBribes()` is not always called by `Voter.distribute()`

Submitted by unforgiven

[Voter.sol#L315-L329]

`Voter.distribute()` calls `Gauge.deliverBribes()` if `claimable[_gauge] / DURATION > 0` was `True` and `claimable[_gauge]` shows `base` token rewards for `gauge`. `Gauge.deliverBribes()` calls `Bribe.deliverReward()` which transfers the rewards to `Gauge`. so for `Bribe` rewards to been transferred and distributed `claimable[_gauge] / DURATION > 0` most be `True` and if there was some new rewards in `Bribe` and that condition is not `True` then those rewards will stuck in `Bribe` contract.

Proof of Concept

This is `Voter.distribute()` code:

```
    function distribute(address _gauge) public lock {
        require(isAlive[_gauge]); // killed gauges cannot distri
        uint dayCalc = block.timestamp % (7 days);
        require((dayCalc < BRIBE_LAG) || (dayCalc > (DURATION +
        IMinter(minter).update_period();
        _updateFor(_gauge);
        uint _claimable = claimable[_gauge];
        if (_claimable > IGauge(_gauge).left(base) && _claimable
            claimable[_gauge] = 0;
            IGauge(_gauge).notifyRewardAmount(base, _claimable);
            emit DistributeReward(msg.sender, _gauge, _claimable
            // distribute bribes & fees too
            IGauge(_gauge).deliverBribes();
        }
    }
```

As you can see `IGauge(_gauge).deliverBribes();` is inside and `if` which checks `_claimable / DURATION > 0` for that `Gauge`. `Bribe` is transferring rewards in `Bribe.deliverReward()` which is called by `Gauge.deliverBribes()` and it is called by `Voter.distribute()` if some `base` token claimable amount of that `Gauge` is bigger than `DURATION`, so if this condition isn't true, then rewards in `Bribe` for that epoch will stuck in `Bribe`.
`Voter.distribute()` can be called multiple times in each time `claimable[_gauge]` will set to `0` so if `Bribe` rewards was after that call then those rewards will stuck in `Bribe` for sure.

Tools Used

VIM

Recommended Mitigation Steps

Move the line `IGauge(_gauge).deliverBribes();` out of `If` in `Voter.distribute()`.

[pooltypes (Velodrome) disputed]

[Alex the Entreprenerd (judge) invalidated and commented]:

> It seems to me like the finding is invalid.

> The check will check if the gauge has any rewards left (via `IGauge(_gauge).left(base)`) (*claimable > IGauge(*gauge).left(base) && _claimable / DURATION > 0) {

> And the `_claimable / DURATION` is checking that the amount is more than a dust amount, as in there's more than 1 wei of token per second.

> Given the information that I have, in lack of a coded POC that shows and quantifies the amount of tokens stuck, I believe the finding to be invalid.

[Alex the Entreprenerd (judge) set severity to Medium and commented]:

> After further review of various findings from the contest, another warden has submitted a convincing POC that shows that this indeed is an enactable vulnerability.

> However, in this finding the warden hasn't shown how the Bribe rewards could be lost.

> Because of that, I think Medium Severity to be more appropriate.

[M-13] Bribe Rewards Not Collected In Current Period Will Be Lost Forever

Submitted by xiaoming90

It was observed that if the bribe rewards are not collected in the current period, they will not be accrued to future epoch, and they will be lost forever.

Proof-of-Concept

When a briber adds in a new bribe reward, the reward information are stored in the `Bribe.tokenRewardsPerEpoch` mapping. This mapping stored the number of rewards tokens per epoch.

Assume that a briber adds 100 DAI token as bribe reward in epoch 0 (current epoch), they will be stored in:

```
tokenRewardsPerEpoch[DAI][Epoch0+1(Bribe Lag)] = 100 DAI tokens
```

Bribe.sol#L41

```
    function notifyRewardAmount(address token, uint amount) external
        require(amount > 0);
        if (!isReward[token]) {
            require(rewards.length < MAX_REWARD_TOKENS, "too many re
        }
        // bribes kick in at the start of next bribe period
        uint adjustedTstamp = getEpochStart(block.timestamp);
        uint epochRewards = tokenRewardsPerEpoch[token][adjustedTsta
```

```
        _safeTransferFrom(token, msg.sender, address(this), amount);
        tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards +

        if (!isReward[token]) {
            isReward[token] = true;
            rewards.push(token);
            IGauge(gauge).addBribeRewardToken(token);
        }

        emit NotifyReward(msg.sender, token, adjustedTstamp, amount)
    }
```

The `Voter.distribute()` function only allows users to distribute the bribe rewards in the current epoch. Calling the `Voter.distribute` function will in turn trigger the `Gauge.deliverBribes` function.

In the `Gauge.deliverBribes` function, the code `uint bribeStart = block.timestamp - (block.timestamp % (7 days)) + BRIBE_LAG;` will always return the start date of current epoch + BRIBE_LAG (1 day). So, if someone call `Gauge.deliverBribes` in epoch 1, the `bribeStart` variable will be set to the `Epoch 1 + 1 day`. Due to the above method of fetching bribe rewards and the use of per epoch's reward mapping, there is no way to fetch the bribe rewards in the previous epoch(s).

Assume that no one triggered the `Voter.distribute()` in epoch 0, the 100 DAI tokens bribe rewards will be lost forever.

Gauge.sol#L173

```
    uint internal constant BRIBE_LAG = 1 days;

    function deliverBribes() external lock {
        require(msg.sender == voter);
        IBribe sb = IBribe(bribe);
        uint bribeStart = block.timestamp - (block.timestamp % (7 da
        uint numRewards = sb.rewardsListLength();

        for (uint i = 0; i < numRewards; i++) {
            address token = sb.rewards(i);
```

```
            // @audit - This will transfer the bribe reward token fr
            uint epochRewards = sb.deliverReward(token, bribeStart);
            if (epochRewards > 0) {
                    // @audit - Update the reward rate accordingly
                _notifyBribeAmount(token, epochRewards, bribeStart);
            }
        }
    }
```

Recommended Mitigation Steps

Devise a new implementation to allow bribe rewards not collected to be accured to future epochs so that the bribe rewards will not be lost. Consider referencing the Solidy's `Bribe` contract, which support these requirements.

pooltypes (Velodrome) acknowledged and commented:

> Assume that no one triggered the Voter.distribute() in epoch 0
>
> Voter.distribute() will always be called as users will have the economic incentive to.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> Given some other submissions in the contest, I believe this finding to be valid, however, the impact is contingent on:
>
> - Nobody calling `deliverBribes` through the voter
> - the impact is limited to those tokens
>
> For those reasons Medium Severity is more appropriate.

[M-14] Wrong reward distribution in Bribe because `deliverReward()` won't set `tokenRewardsPerEpoch[token][epochStart]` to 0

Submitted by unforgiven, also found by csanuragjain, pcrypt0, and smilingheretic

Bribe.sol#L83-L90

Function `deliverReward()` in `Bribe` contract won't set `tokenRewardsPerEpoch[token][epochStart]` to `0` after transferring rewards. `Gauge.getReward()` calls `Voter.distribute()` which calls `Gauge.deliverBribes()` which calls `Bribe.deliverReward()` . so if `Gauge.getReward()` or `Voter.distribute()` get called multiple times in same epoch then `deliverReward()` will transfer `Bribe` tokens multiple times because it doesn't set `tokenRewardsPerEpoch[token][epochStart]` to `0` after transferring.

Proof of Concept

This is `deliverReward()` code in `Bribe` :

```
function deliverReward(address token, uint epochStart) externa
  require(msg.sender == gauge);
  uint rewardPerEpoch = tokenRewardsPerEpoch[token][epochStart
  if (rewardPerEpoch > 0) {
    _safeTransfer(token, address(gauge), rewardPerEpoch);
  }
  return rewardPerEpoch;
}
```

As you can see it doesn't set `tokenRewardsPerEpoch[token][epochStart]` value to `0` , so if this function get called multiple times it will transfer epoch rewards multiple times (it will use other epoch's rewards tokens).
function `Gauge.deliverBribes()` calls `Bribe.deliverReward()` and `Gauge.deliverBribes()` is called by `Voter.distribute()` if the condition `claimable[_gauge] > DURATION` is `True` . This is those functions codes:

```
function deliverBribes() external lock {
    require(msg.sender == voter);
    IBribe sb = IBribe(bribe);
    uint bribeStart = block.timestamp - (block.timestamp % (
    uint numRewards = sb.rewardsListLength();

    for (uint i = 0; i < numRewards; i++) {
        address token = sb.rewards(i);
        uint epochRewards = sb.deliverReward(token, bribeSta
        if (epochRewards > 0) {
            _notifyBribeAmount(token, epochRewards, bribeSta
```

```
            }
        }
    }


    function distribute(address _gauge) public lock {
        require(isAlive[_gauge]); // killed gauges cannot distri
        uint dayCalc = block.timestamp % (7 days);
        require((dayCalc < BRIBE_LAG) || (dayCalc > (DURATION +
        IMinter(minter).update_period();
        _updateFor(_gauge);
        uint _claimable = claimable[_gauge];
        if (_claimable > IGauge(_gauge).left(base) && _claimable
            claimable[_gauge] = 0;
            IGauge(_gauge).notifyRewardAmount(base, _claimable);
            emit DistributeReward(msg.sender, _gauge, _claimable
            // distribute bribes & fees too
            IGauge(_gauge).deliverBribes();
        }
    }
```

also `Gauge.getReward()` calls `Voter.getReward()`.
condition `claimable[_gauge] > DURATION` in `Voter.distribute()` can be true multiple time in one epoch (`deliverBribes()` would be called multiple times) because `claimable[_gauge]` is based on `index` and `index` increase by `notifyRewardAmount()` in `Voter` anytime.

Tools Used

VIM

Recommended Mitigation Steps

Set `tokenRewardsPerEpoch[token][epochStart]` to `0` in `deliverReward`.

pooltypes (Velodrome) confirmed and commented:

> Thanks, this is an issue we discovered in prod. Issuing a fix soon.

> We're also taking the necessary steps to alert any users who may have funds at risk from this issue.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> `tokenRewardsPerEpoch[token][epochStart]` is indeed not resetted on each claim

> Proving the finding is reliant on `claimable[_gauge] += _share;` which is set in `_updateFor`

> Which is contingent on `index += _ratio;` in `notifyRewardAmount`

> Because `notifyRewardAmount` can be called by anyone, at the cost of an amount that just needs to be greater than or equal to `totalWeight / 1e18` which may be a negligible amount, then indeed `index` can increase, allowing the bribe to be called multiple times in one epoch.

> In terms of impact, because the Bribe contract only allows for an extremely basic, now or in 7 days, type queueing of bribes, the accounting of bribes is mostly unnecessary (as the contract will most of the times be sending all tokens anyway).

> Additionally the loss would amount to an incorrect amount of bribes emitted over 7 days instead of 14.

> For those reasons, I think Medium Severity to be more appropriate.

[M-15] Wrong calculation for the new `rewardRate[token]` can cause some of the late users can not get their rewards

Submitted by WatchPug

```
uint bribeStart = block.timestamp - (block.timestamp % (7 days))
uint adjustedTstamp = block.timestamp < bribeStart ? bribeStart
if (rewardRate[token] == 0) _writeRewardPerTokenCheckpoint(toker
(rewardPerTokenStored[token], lastUpdateTime[token]) = _updateRe
_claimFees();

if (block.timestamp >= periodFinish[token]) {
    _safeTransferFrom(token, msg.sender, address(this), amount);
```

```
        rewardRate[token] = amount / DURATION;
    } else {
        uint _remaining = periodFinish[token] - block.timestamp;
        uint _left = _remaining * rewardRate[token];
        require(amount > _left);
        _safeTransferFrom(token, msg.sender, address(this), amount);
        rewardRate[token] = (amount + _left) / DURATION;
    }
```

In `Gauge.sol#notifyRewardAmount()`, the updated rewardRate for the token: `rewardRate[token]` is calculated based on the newly added amount of tokens (`amount`), the remaining amount of existing rewards (`_left`), and the `DURATION`.

While the DURATION is `5 days`, the period from the current time to `periodFinish[token]` is much longer.

`rewardPerToken()` is calculated based on the current time, `lastUpdateTime[token]`, and `rewardRate[token]`.

Gauge.sol#L375-L380

```
    function rewardPerToken(address token) public view returns (uint
        if (derivedSupply == 0) {
            return rewardPerTokenStored[token];
        }
        return rewardPerTokenStored[token] + ((lastTimeRewardApplica
    }
```

`lastUpdateTime[token]` will frequently be updated to the current timestamp by `_updateRewardForAllTokens()`.

See: Gauge.sol#L460-L469

As a result, `rewardPerToken()` can be much higher than expected, which makes the total amount of reward tokens less than the total amount of rewards accumulated by all the users.

This makes the users who claim the rewards later unable to retrieve their rewards as the balance can be insufficient.

Proof of Concept

1. Alice and Bob both deposited `1,000` stake token to Gauge at `1653091200` (May 21 2022 00:00:00 GMT+0000)
2. Admin called `notifyRewardAmount()` add 1,000 DAI at `1653100000` (May 21 2022 02:26:40 GMT+0000)
3. `bribeStart` = 1653004800 (May 20 2022 00:00:00 GMT+0000)
4. `adjustedTstamp` = 1653609600 (May 27 2022 00:00:00 GMT+0000)
5. `periodFinish[DAI]` = 1654041600 (Jun 01 2022 00:00:00 GMT+0000)
6. `lastUpdateTime[DAI]` = 1653100000
7. `rewardRate[DAI]` = 1,000 * 1e18 / 432000 = 2314814814814815
8. Alice withdrawn and `getReward()` at `1654041800` Jun 01 2022 00:03:20 GMT+0000, get ~1,000 DAI
9. `rewardPerTokenStored[DAI]` = ~1e18
10. Bob tried to `getReward()`, the transaction will revert due to insufficient balance.

Recommended Mitigation Steps

Consider calculating `rewardRate` base on timeUntilNextPeriodFinish to next period finish:

```
    uint nextPeriodFinish = adjustedTstamp + DURATION;
    uint timeUntilNextPeriodFinish = nextPeriodFinish - block.timest
    if (block.timestamp >= periodFinish[token]) {
        _safeTransferFrom(token, msg.sender, address(this), amount);
        rewardRate[token] = amount / timeUntilNextPeriodFinish;
    } else {
        uint _remaining = periodFinish[token] - block.timestamp;
        uint _left = _remaining * rewardRate[token];
        require(amount > _left);
        _safeTransferFrom(token, msg.sender, address(this), amount);
        rewardRate[token] = (amount + _left) / timeUntilNextPeriodFi
    }
```

```
    require(rewardRate[token] > 0);
    uint balance = IERC20(token).balanceOf(address(this));
    require(rewardRate[token] <= balance / timeUntilNextPeriodFinish
    periodFinish[token] = nextPeriodFinish;
```

[pooltypes (Velodrome) acknowledged]

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]:

> The warden has shown how a desynch between `periodFinish` and `DURATION` can cause rewards to be distributed faster than a reward period, leaving the last claimers unable to claim.

> While the loss in this scenario, to those claimers can be viewed as total, in reality those people will be able to claim on the next rewards round (unless no more tokens will be minted to that gauge, which is highly unlikely but possible).

> For that reason, I believe Medium Severity to be more appropriate, as this will be a temporary loss of yield which for some people may be permanent.

[M-16] Wrong `DOMAIN_TYPEHASH` definition

Submitted by rotcivegaf

Broke the [EIP 712], and the `delegateBySig` function.

Proof of Concept

In the build of the `DOMAIN TYPEHASH` the `string version` is forgotten, but the `delegateBySig` function, build the `domainSeparator` with the `string version`.

Some contract or dapp/backend could building the `DOMAIN_TYPEHASH` with *"rigth"* struct(include the `version`) and try to use the `delegateBySig` function but this function will revert in the L1378 with the message `"VotingEscrow::delegateBySig: invalid signature"` because the expect `DOMAIN_TYPEHASH` in the `VotingEscrow.sol` contract was built with the *"wrong"* struct.

Recommended Mitigation Steps

Acording the EIP 712, in the Definition of domainSeparator:

- "`string version` *the current major version of the signing domain. Signatures from different versions are not compatible*"

Add `string version`, to the `EIP712Domain` string, L1106:

```
bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domai
```

Other Recommendation

Build the `domainSeparator` L1362 in the constructor to save gas and clarify/clean code

1. Remove the `bytes32 public constant DOMAIN_TYPEHASH` L1106
2. Add `bytes32 private immutable DOMAIN_SEPARATOR` as a contract VAR
3. Assign the `DOMAIN_SEPARATOR` in the constructor
4. Use the `DOMAIN_SEPARATOR` in `delegateBySig` function when buil the `bytes32 digest`

```
contract VotingEscrow is IERC721, IERC721Metadata, IVotes {

    ...

    /* DELETE THIS LINE
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712I
    */
    bytes32 private immutable DOMAIN_SEPARATOR;

    ...

    constructor(address token_addr) {
        bytes32 domainTypeHash = keccak256(
            "EIP712Domain(string name,string version,uint256 cha
        );
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
```

```
                domainTypeHash,
                keccak256(bytes(name)),
                keccak256(bytes(version)),
                block.chainid,
                address(this)
            )
        );

    ...

    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public {
        bytes32 structHash = keccak256(
            abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, ex
        );
        bytes32 digest = keccak256(
            abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR, struc
        );

    ...
```

[pooltypes (Velodrome) disputed]

[Alex the Entreprenerd (judge) commented]:

> The warden has shown how the code has broken EIP-712. I do not expect any security risk, and I assume wallets will still pick up the code; however, because the code breaks the standard, in line with another finding I've judged in this contest, I think Medium severity to be appropriate.

[M-17] WeVE (FTM) may be lost forever if redemption process is failed

Submitted by Chom

RedemptionSender.sol#L28-L51
RedemptionReceiver.sol#L72-L105

WeVE (FTM) may be lost forever if redemption process is failed.

Redemption process is likely to be failed if

- (redeemedWEVE += amountWEVE) > eligibleWEVE
- Not enough USDC or VELO in the contract

The case that redeem more than eligible can't be fixed because eligibleWEVE is hardcoded on contract initialization.

This mean that if there are any mistake for example LayerZero slow down and user try to repeatedly redeem their WeVE, user will lose their WeVE token forever due to contract always reverted in the destination chain due to the reason that user has redeemed more than eligible.

Proof of Concept

1. User redeem WeVE in fantom chain using redeemWEVE function in RedemptionSender contract.
2. LayerZero slow but user think it is failed. (But it is just slow)
3. User repeat process 1 again
4. LayerZero call lzReceive in RedemptionReceiver contract on Optimism chain for the first time it's success. USDC + VELO is redeemed as intended.
5. LayerZero call lzReceive in RedemptionReceiver contract on Optimism chain again due to repeated transaction in step 3. But this time, user has exceeded her redeem limit. Caused lzReceive call to revert with reason "cannot redeem more than eligible". **But doesn't refund WeVE to the user**

```
require(
    (redeemedWEVE += amountWEVE) <= eligibleWEVE,
    "cannot redeem more than eligible"
);
```

6. User FUD Velodrome and file a lawsuit against Velodrome.

Recommended Mitigation Steps

- In RedemptionReceiver, Wrap lzReceive into another function and perform try catch on new lzReceive function to call old wrapped lzReceive function and on revert add refund amount to that user.
- Write refund lzReceive handler on RedemptionSender.
- Create a new refund function in RedemptionReceiver. When user call, it will send layerzero message back to lzReceive function in RedemptionSender contract on Fantom.

pooltypes (Velodrome) disputed and disagreed with severity

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I believe the finding to have validity exclusively on the basis of the fact that a user may burn their WeVe and reach cap on the receiving chain, getting nothing out of it.

> Because that's contingent on reaching cap, the loss will be limited to the capped amount. For that reason, I think Medium Severity to be more appropriate.

Low Risk and Non-Critical Issues

For this contest, 50 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **IIIIIII** received the top score from the judge.

The following wardens also submitted reports: **0xNazgul**, **hake**, **0x1f8b**, **cryptphi**, **minhquanym**, **Funen**, **rotcivegaf**, **Dravee**, **hansfriese**, **sashik_eth**, **0xNineDec**, **csanuragjain**, **gzeon**, **Hawkeye**, **_Adam**, **robee**, **GimelSec**, **WatchPug**, **teddav**, **berndartmueller**, **horsefacts**, **Nethermind**, **hyh**, **p_crypt0**, **TerrierLover**, **djxploit**, **xiaoming90**, **jayjonah8**, **unforgiven**, **0x52**, **catchup**, **pauliax**, **cccz**, **asutorufos**, **c3phas**, **sorrynotsorry**, **simon135**, **MaratCerby**, **delfin454000**, **BouSalman**, **oyc_109**, **Picodes**, **sach1r0**, **Chom**, **fatherOfBlocks**, **AlleyCat**, **CertoraInc**, **RoiEvenHaim**, *and* **SooYa**.

Summary

Low Risk Issues

	Issue	Instances
L-01	`Math.max(<x>,0)` used with `int` cast to `uint`	4
L-02	Front-runable initializer	1
L-03	`require()` should be used instead of `assert()`	18
L-04	`_safeMint()` should be used rather than `_mint()` wherever possible	1
L-05	Missing checks for `address(0x0)` when assigning values to `address` state variables	33

Total: 57 instances over 5 issues

Non-critical Issues

	Issue	Instances
N-01	`approveMax` variable causes problems	2
N-02	Only a billion checkpoints available	1
N-03	Open TODOs	4
N-04	Use two-phase ownership transfers	4
N-05	Avoid the use of sensitive terms	3
N-06	`require()` / `revert()` statements should have descriptive reason strings	93
N-07	`public` functions not called by the contract should be declared `external` instead	9
N-08	`constant` s should be defined rather than using magic numbers	95
N-09	Redundant cast	4

	Issue	Instances
N-10	Numeric values having to do with time should use time units for readability	7
N-11	Large multiples of ten should use scientific notation (e.g. `1e6`) rather than decimal literals (e.g. `1000000`), for readability	3
N-12	Missing event for critical parameter change	16
N-13	Variable names that consist of all capital letters should be reserved for `constant` / `immutable` variables	1
N-14	Typos	7
N-15	File is missing NatSpec	13
N-16	Event is missing `indexed` fields	31
N-17	Not using the named return variables anywhere in the function is confusing	12

Total: 305 instances over 17 issues

[L-01] `Math.max(<x>,0)` used with `int` cast to `uint`

The code casts an `int` to a `uint` before passing it to `Math.max()`. It seems as though the `Math.max()` call is attempting to prevent values from being negative, but since the `int` is being cast to `uint`, the value will never be negative, and instead will overflow if either the multiplication involving the slope and timestamp is positive. I wasn't able to find a scenario where this is the case, but this seems very dangerous, and the `Math.max()` call is sending misleading signals, so I suggest moving it to inside the cast to `uint`

There are 4 instances of this issue:

```
File: contracts/contracts/RewardsDistributor.sol   #1

139:         return Math.max(uint(int256(pt.bias - pt.slope * (i
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L139

```
File: contracts/contracts/RewardsDistributor.sol   #2

158:                 ve_supply[t] = Math.max(uint(int256(pt.bias
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L158

```
File: contracts/contracts/RewardsDistributor.sol   #3

208:                 uint balance_of = Math.max(uint(int256(old_
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L208

```
File: contracts/contracts/RewardsDistributor.sol   #4

265:                 uint balance_of = Math.max(uint(int256(old_
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L265

[L-02] Front-runable initializer

There is nothing preventing another account from calling the initializer before the contract owner. In the best case, the owner is forced to waste gas and re-deploy. In the worst case, the owner does not notice that his/her call reverts, and everyone starts using a contract under the control of an attacker

There is 1 instance of this issue:

```
File: contracts/contracts/Bribe.sol   #1

30      function setGauge(address _gauge) external {
31        require(gauge == address(0), "gauge already set");
32        gauge = _gauge;
33:     }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Bribe.sol#L30-L33

[L-03] `require()` should be used instead of `assert()`

Prior to solidity version 0.8.0, hitting an assert consumes the **remainder of the transaction's available gas** rather than returning it, as `require()` / `revert()` do. `assert()` should be avoided even past solidity version 0.8.0 as its documentation states that "The assert function creates an error of type Panic(uint256). ... Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix".

There are 18 instances of this issue:

```
File: contracts/contracts/Router.sol

36:           assert(msg.sender == address(weth)); // only accep

181:                assert(amountAOptimal <= amountADesired);

227:          assert(weth.transfer(pair, amountETH));

373:          assert(weth.transfer(pairFor(routes[0].from, route
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Router.sol#L36

```
File: contracts/contracts/VotingEscrow.sol
```

```
262:          assert(_operator != msg.sender);

272:          assert(idToOwner[_tokenId] == _owner);

447:          assert(idToOwner[_tokenId] == address(0));

464:          assert(_to != address(0));

508:          assert(idToOwner[_tokenId] == _from);

748:              assert(IERC20(token).transferFrom(from, addres

815:          assert(_isApprovedOrOwner(msg.sender, _tokenId));

819:          assert(_value > 0); // dev: need non-zero value

829:          assert(_isApprovedOrOwner(msg.sender, _tokenId));

845:          assert(_isApprovedOrOwner(msg.sender, _tokenId));

861:          assert(IERC20(token).transfer(msg.sender, value));

937:          assert(_block <= block.number);

991:          assert(_block <= block.number);
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L262

```
File: contracts/contracts/RewardsDistributor.sol

98:           assert(msg.sender == depositor);
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L98

[L-04] `_safeMint()` should be used rather than `_mint()` wherever possible

`_mint()` is **discouraged** in favor of `_safeMint()` which ensures that the recipient is either an EOA or implements `IERC721Receiver`. Both **OpenZeppelin** and **solmate** have versions of this function

There is 1 instance of this issue:

```
File: contracts/contracts/VotingEscrow.sol   #1

791:          _mint(_to, _tokenId);
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L791

[L-05] Missing checks for `address(0x0)` when assigning values to `address` state variables

There are 33 instances of this issue. For in-depth details, see the warden's **full report**.

[N-01] `approveMax` variable causes problems

The `approveMax` variable controls whether the liquidity value is used, or the maximum is used. If the result of that check doesn't match what was provided during signature generation, the `permit()` call will fail, so in the best case the variable has no effect. In the worst case it leads to incorrect state handling

There are 2 instances of this issue:

```
File: contracts/contracts/Router.sol   #1

290             uint value = approveMax ? type(uint).max : liqu
291:            IPair(pair).permit(msg.sender, address(this), v
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/cont

racts/Router.sol#L290-L291

```
File: contracts/contracts/Router.sol   #2

308           uint value = approveMax ? type(uint).max : liquidit
309:          IPair(pair).permit(msg.sender, address(this), value
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Router.sol#L308-L309

[N-02] Only a billion checkpoints available

A user can only have a billion checkpoints which, if the user is a DAO, may cause issues down the line, especially if the last checkpoint involved delegating and can thereafter not be undone

There is 1 instance of this issue:

```
File: contracts/contracts/VotingEscrow.sol   #1

535:      mapping(uint => Point[1000000000]) public user_point_hi
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L535

[N-03] Open TODOs

Code architecture, incentives, and error handling/reporting questions/issues should be resolved before deployment

There are 4 instances of this issue:

```
File: contracts/contracts/VotingEscrow.sol   #1
```

```
314:          // TODO delegates
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L314

```
File: contracts/contracts/VotingEscrow.sol   #2

465:          // TODO add delegates
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L465

```
File: contracts/contracts/VotingEscrow.sol   #3

524:          // TODO add delegates
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L524

```
File: contracts/contracts/VelodromeLibrary.sol   #4

9:        IRouter internal immutable router; // TODO make modifi
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VelodromeLibrary.sol#L9

[N-04] Use two-phase ownership transfers

Consider adding a two-phase transfer, where the current owner nominates the next owner, and the next owner has to call `accept*()` to become the new owner. This prevents passing the ownership to an account that is unable to use it.

There are 4 instances of this issue:

```
File: contracts/contracts/factories/GaugeFactory.sol   #1

17        function setTeam(address _team) external {
18            require(msg.sender == team);
19            team = _team;
20:       }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/factories/GaugeFactory.sol#L17-L20

```
File: contracts/contracts/VeloGovernor.sol   #2

39        function setTeam(address newTeam) external {
40            require(msg.sender == team, "not team");
41            team = newTeam;
42:       }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VeloGovernor.sol#L39-L42

```
File: contracts/contracts/Voter.sol   #3

82        function setGovernor(address _governor) public {
83            require(msg.sender == governor);
84            governor = _governor;
85:       }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L82-L85

```
File: contracts/contracts/Voter.sol   #4
```

```
87        function setEmergencyCouncil(address _council) public {
88            require(msg.sender == emergencyCouncil);
89            emergencyCouncil = _council;
90:       }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L87-L90

[N-05] Avoid the use of sensitive terms

Use alternative variants, e.g. allowlist/denylist instead of whitelist/blacklist

There are 3 instances of this issue:

```
File: contracts/contracts/Voter.sol   #1

38:       mapping(address => bool) public isWhitelisted;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L38

```
File: contracts/contracts/Voter.sol   #2

52:       event Whitelisted(address indexed whitelister, address
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L52

```
File: contracts/contracts/Voter.sol   #3

178:      function whitelist(address _token) public {
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L178

[N-06] `require()` / `revert()` statements should have descriptive reason strings

There are 93 instances of this issue. For in-depth details, see the warden's full report.

[N-07] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

There are 9 instances of this issue:

```
File: contracts/contracts/redeem/RedemptionSender.sol

28          function redeemWEVE(
29              uint256 amount,
30              address zroPaymentAddress,
31:             bytes memory zroTransactionParams
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/redeem/RedemptionSender.sol#L28-L31

```
File: contracts/contracts/Gauge.sol

163:        function getVotingStage(uint timestamp) public pure re
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Gauge.sol#L163

File: contracts/contracts/VotingEscrow.sol

```
1184      function getPastVotes(address account, uint timestamp)
1185          public
1186          view
1187:         returns (uint)

1349:     function delegate(address delegatee) public {

1354      function delegateBySig(
1355          address delegatee,
1356          uint nonce,
1357          uint expiry,
1358          uint8 v,
1359          bytes32 r,
1360:         bytes32 s
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L1184-L1187

File: contracts/contracts/factories/PairFactory.sol

```
76:       function getFee(bool _stable) public view returns(uint
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/factories/PairFactory.sol#L76

File: contracts/contracts/Voter.sol

```
82:       function setGovernor(address _governor) public {

87:       function setEmergencyCouncil(address _council) public

178:      function whitelist(address _token) public {
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/cont

racts/Voter.sol#L82

[N-08] `constant`s should be defined rather than using magic numbers

There are 95 instances of this issue. For in-depth details, see the warden's full report.

[N-09] Redundant cast

The type of the variable is the same as the type to which the variable is being cast

There are 4 instances of this issue:

```
File: contracts/contracts/Bribe.sol   #1

/// @audit address(gauge)
87:         _safeTransfer(token, address(gauge), rewardPerEpoch)
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Bribe.sol#L87

```
File: contracts/contracts/Voter.sol   #2

/// @audit uint256(_totalWeight)
118:          totalWeight -= uint256(_totalWeight);
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L118

```
File: contracts/contracts/Voter.sol   #3

/// @audit uint256(_totalWeight)
168:          totalWeight += uint256(_totalWeight);
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L168

```
File: contracts/contracts/Voter.sol   #4

/// @audit uint256(_usedWeight)
169:           usedWeights[_tokenId] = uint256(_usedWeight);
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L169

[N-10] Numeric values having to do with time should use time units for readability

There are **units** for seconds, minutes, hours, days, and weeks

There are 7 instances of this issue:

```
File: contracts/contracts/Minter.sol

/// @audit 86400
14:       uint internal constant WEEK = 86400 * 7; // allows mir

/// @audit 15000000e18
22:       uint public weekly = 15000000e18;

/// @audit 86400
24:       uint internal constant LOCK = 86400 * 7 * 52 * 4;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Minter.sol#L14

```
File: contracts/contracts/VotingEscrow.sol

/// @audit 86400
```

```
542:      uint internal constant MAXTIME = 4 * 365 * 86400;

/// @audit 86400
543:      int128 internal constant iMAXTIME = 4 * 365 * 86400;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L542

```
File: contracts/contracts/RewardsDistributor.sol

/// @audit 86400
30:       uint constant WEEK = 7 * 86400;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L30

```
File: contracts/contracts/Pair.sol

/// @audit 1800
45:       uint constant periodSize = 1800;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Pair.sol#L45

[N-11] Large multiples of ten should use scientific notation (e.g. `1e6`) rather than decimal literals (e.g. `1000000`), for readability

There are 3 instances of this issue:

```
File: contracts/contracts/VotingEscrow.sol   #1

535:      mapping(uint => Point[1000000000]) public user_point_h
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L535

```
File: contracts/contracts/RewardsDistributor.sol   #2

38:       uint[1000000000000000] public tokens_per_week;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L38

```
File: contracts/contracts/RewardsDistributor.sol   #3

44:       uint[1000000000000000] public ve_supply;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L44

[N-12] Missing event for critical parameter change

There are 16 instances of this issue:

```
File: contracts/contracts/Minter.sol

64        function setTeam(address _team) external {
65            require(msg.sender == team, "not team");
66            pendingTeam = _team;
67:       }

74        function setTeamRate(uint _teamRate) external {
75            require(msg.sender == team, "not team");
76            require(_teamRate <= MAX_TEAM_RATE, "rate too high
77            teamRate = _teamRate;
78:       }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Minter.sol#L64-L67

```
File: contracts/contracts/VotingEscrow.sol

1059      function setVoter(address _voter) external {
1060          require(msg.sender == voter);
1061          voter = _voter;
1062:     }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L1059-L1062

```
File: contracts/contracts/VeloGovernor.sol

39        function setTeam(address newTeam) external {
40            require(msg.sender == team, "not team");
41            team = newTeam;
42:       }

44        function setProposalNumerator(uint256 numerator) exter
45            require(msg.sender == team, "not team");
46            require(numerator <= MAX_PROPOSAL_NUMERATOR, "nume
47            proposalNumerator = numerator;
48:       }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VeloGovernor.sol#L39-L42

```
File: contracts/contracts/RewardsDistributor.sol

318       function setDepositor(address _depositor) external {
319           require(msg.sender == depositor);
320           depositor = _depositor;
321:      }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol#L318-L321

File: contracts/contracts/Velo.sol

```
26          function setMinter(address _minter) external {
27              require(msg.sender == minter);
28              minter = _minter;
29:         }

31          function setRedemptionReceiver(address _receiver) exte
32              require(msg.sender == minter);
33              redemptionReceiver = _receiver;
34:         }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Velo.sol#L26-L29

File: contracts/contracts/Bribe.sol

```
30        function setGauge(address _gauge) external {
31          require(gauge == address(0), "gauge already set");
32          gauge = _gauge;
33:       }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Bribe.sol#L30-L33

File: contracts/contracts/factories/GaugeFactory.sol

```
17          function setTeam(address _team) external {
18              require(msg.sender == team);
19              team = _team;
20:         }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/factories/GaugeFactory.sol#L17-L20

File: contracts/contracts/factories/PairFactory.sol

```
40          function setPauser(address _pauser) external {
41              require(msg.sender == pauser);
42              pendingPauser = _pauser;
43:         }

50          function setPause(bool _state) external {
51              require(msg.sender == pauser);
52              isPaused = _state;
53:         }

55          function setFeeManager(address _feeManager) external {
56              require(msg.sender == feeManager, 'not fee manager
57              pendingFeeManager = _feeManager;
58:         }

65          function setFee(bool _stable, uint256 _fee) external {
66              require(msg.sender == feeManager, 'not fee manager
67              require(_fee <= MAX_FEE, 'fee too high');
68              require(_fee != 0, 'fee must be nonzero');
69              if (_stable) {
70                  stableFee = _fee;
71              } else {
72                  volatileFee = _fee;
73              }
74:         }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/factories/PairFactory.sol#L40-L43

File: contracts/contracts/Voter.sol

```
82          function setGovernor(address _governor) public {
83              require(msg.sender == governor);
84              governor = _governor;
85:         }
```

```
87          function setEmergencyCouncil(address _council) public
88              require(msg.sender == emergencyCouncil);
89              emergencyCouncil = _council;
90:         }
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol#L82-L85

[N-13] Variable names that consist of all capital letters should be reserved for `constant` / `immutable` variables

If the variable needs to be different based on which class it comes from, a `view` / `pure` *function* should be used instead (e.g. like this).

There is 1 instance of this issue:

```
File: contracts/contracts/Pair.sol   #1

25:         bytes32 internal DOMAIN_SEPARATOR;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Pair.sol#L25

[N-14] Typos

There are 7 instances of this issue:

```
File: contracts/contracts/redeem/RedemptionReceiver.sol

/// @audit FTM
14:         uint16 public immutable fantomChainId; // 12 for FTM,
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/cont

racts/redeem/RedemptionReceiver.sol#L14

```
File: contracts/contracts/VotingEscrow.sol

/// @audit blocktimes
38:         * and per block could be fairly bad b/c Ethereum char

/// @audit Exeute
295:      /// @dev Exeute transfer of a NFT.

/// @audit Pevious
575:      /// @param old_locked Pevious locked amount / end lock
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L38

```
File: contracts/contracts/PairFees.sol

/// @audit localy
10:       address internal immutable token0; // token0 of pair,

/// @audit localy
11:       address internal immutable token1; // Token1 of pair,
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/PairFees.sol#L10

```
File: contracts/contracts/Pair.sol

/// @audit obervations
37:       // Structure to capture time period obervations every
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Pair.sol#L37

[N-15] File is missing NatSpec

There are 13 instances of this issue:

File: contracts/contracts/Minter.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Minter.sol

File: contracts/contracts/Router.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Router.sol

File: contracts/contracts/VeloGovernor.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VeloGovernor.sol

File: contracts/contracts/VelodromeLibrary.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VelodromeLibrary.sol

File: contracts/contracts/RewardsDistributor.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/RewardsDistributor.sol

File: contracts/contracts/PairFees.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/PairFees.sol

File: contracts/contracts/Velo.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Velo.sol

File: contracts/contracts/Bribe.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Bribe.sol

File: contracts/contracts/factories/GaugeFactory.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/factories/GaugeFactory.sol

File: contracts/contracts/factories/BribeFactory.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/factories/BribeFactory.sol

File: contracts/contracts/factories/PairFactory.sol

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/factories/PairFactory.sol

```
File: contracts/contracts/Pair.sol
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Pair.sol

```
File: contracts/contracts/Voter.sol
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Voter.sol

[N-16] Event is missing `indexed` fields

Each `event` should use three `indexed` fields if there are three or more fields

There are 31 instances of this issue. For in-depth details, see the warden's full report.

[N-17] Not using the named return variables anywhere in the function is confusing

Consider changing the variable to be an unnamed one

There are 12 instances of this issue:

```
File: contracts/contracts/Router.sol

/// @audit amount
71:       function getAmountOut(uint amountIn, address tokenIn,

/// @audit stable
71:       function getAmountOut(uint amountIn, address tokenIn,
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Router.sol#L71

```
File: contracts/contracts/Gauge.sol

/// @audit claimed0
131:      function claimFees() external lock returns (uint claim

/// @audit claimed1
131:      function claimFees() external lock returns (uint claim
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Gauge.sol#L131

```
File: contracts/contracts/Pair.sol

/// @audit dec0
125:      function metadata() external view returns (uint dec0,

/// @audit dec1
125:      function metadata() external view returns (uint dec0,

/// @audit r0
125:      function metadata() external view returns (uint dec0,

/// @audit r1
125:      function metadata() external view returns (uint dec0,

/// @audit st
125:      function metadata() external view returns (uint dec0,

/// @audit t0
125:      function metadata() external view returns (uint dec0,

/// @audit t1
125:      function metadata() external view returns (uint dec0,

/// @audit amountOut
254:      function quote(address tokenIn, uint amountIn, uint gr
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Pair.sol#L125

Alex the Entreprenerd (judge) commented:

> All in all a really thorough report, which would benefit by mentioning a finding once, and then listing all other occurrences, especially when the code is pretty much the same.

> That said, one of the best reports.

> *(Note: See original submission for judge's full commentary.)*

Gas Optimizations

For this contest, 51 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: MiloTruck, 0x1f8b, reassor, rotcivegaf, sashik_eth, supernova, gzeon, 0x4non, Waze, Dravee, MaratCerby, saian, simon135, MadWookie, GimelSec, hansfriese, _Adam, 0xkatana, c3phas, UnusualTurtle, TerrierLover, 0xNazgul, Funen, ElKu, asutorufos, catchup, DavidGialdi, delfin454000, djxploit, fatherOfBlocks, oyc_109, pauliax, rfa, sach1r0, Tomio, TomJ, WatchPug, z3s, Chom, Deivitto, Fitraldys, orion, Picodes, teddav, hake, horsefacts, 0xf15ers, csanuragjain, ControlCplusControlV, *and* Randyyy.

Summary

	Issue	Instances
G-01	Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate	8
G-02	State variables only set in the constructor should be declared `immutable`	5
G-	State variables can be packed into fewer storage slots	1

	Issue	Instances
03		
G-04	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	13
G-05	State variables should be cached in stack variables rather than re-reading them from storage	39
G-06	Multiple accesses of a mapping should use a local variable cache	34
G-07	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	21
G-08	`internal` functions only called once can be inlined to save gas	25
G-09	`<array>.length` should not be looked up in every loop of a `for` -loop	17
G-10	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops	41
G-11	`require()` / `revert()` strings longer than 32 bytes cost extra gas	14
G-12	`keccak256()` should only need to be called on a specific string literal once	1
G-13	Using `bool` s for storage incurs overhead	14
G-14	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	6
G-15	It costs more gas to initialize variables to zero than to let the default of zero be applied	68
G-16	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	37
G-17	Splitting `require()` statements that use `&&` saves gas	17
G-18	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	45
G-19	`abi.encode()` is less efficient than `abi.encodePacked()`	1
G-20	Using `private` rather than `public` for constants, saves gas	18

	Issue	Instances
G-21	Duplicated `require()` / `revert()` checks should be refactored to a modifier or function	30
G-22	Division by two should use bit shifting	9
G-23	Stack variable used as a cheaper cache for a state variable is only used once	1
G-24	`require()` or `revert()` statements that check input arguments should be at the top of the function	13
G-25	Use custom errors rather than `revert()` / `require()` strings to save deployment gas	86

Total: 564 instances over 25 issues

[G-01] Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (**20000 gas**) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations.

There are 8 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, see the warden's full report.)

[G-02] State variables only set in the constructor should be declared `immutable`

Avoids a Gsset (**20000 gas**) in the constructor, and replaces each Gwarmacces (**100 gas**) with a `PUSH32` (**3 gas**).

There are 5 instances of this issue.

[G-03] State variables can be packed into fewer storage slots

If variables occupying the same slot are both written the same function or by the constructor, avoids a separate Gsset (**20000 gas**). Reads of the variables can also be cheaper

There is 1 instance of this issue:

```
File: contracts/contracts/VotingEscrow.sol   #1

/// @audit Variable ordering with 23 slots instead of the currer
/// @audit  mapping(32):point_history, mapping(32):supportedInte
67:       address public voter;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L67

[G-04] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one

There are 13 instances of this issue.

[G-05] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 39 instances of this issue.

[G-06] Multiple accesses of a mapping should use a local variable cache

The instances below point to the second+ access of a value inside a mapping, within a function. Caching a mapping's value in a local `storage` variable when the value is accessed multiple times, saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations.

There are 34 instances of this issue.

[G-07] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

There are 21 instances of this issue.

[G-08] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 25 instances of this issue.

[G-09] `<array>.length` should not be looked up in every loop of a `for` -loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 17 instances of this issue.

[G-10] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** per loop

There are 41 instances of this issue.

[G-11] `require()` / `revert()` strings longer than 32 bytes cost extra gas

There are 14 instances of this issue.

[G-12] `keccak256()` should only need to be called on a specific string literal once

It should be saved to an immutable variable, and the variable used instead. If the hash is being used as a part of a function selector, the cast to `bytes4` should also only be done once

There is 1 instance of this issue:

```
File: contracts/contracts/Pair.sol   #1
```

```
470:                 keccak256('EIP712Domain(string name,string
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/Pair.sol#L470

[G-13] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27 Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (**20000 gas**) when changing from ‘false’ to ‘true’, after having been ‘true’ in the past

There are 14 instances of this issue.

[G-14] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves 6 gas per instance

There are 6 instances of this issue.

[G-15] It costs more gas to initialize variables to zero than to let the default of zero be applied

There are 68 instances of this issue.

[G-16] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **6 gas per loop**

There are 37 instances of this issue.

[G-17] Splitting `require()` statements that use `&&` saves gas

See **this issue** which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper

There are 17 instances of this issue.

[G-18] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

There are 45 instances of this issue.

[G-19] `abi.encode()` is less efficient than `abi.encodePacked()`

There is 1 instance of this issue:

```
File: contracts/contracts/redeem/RedemptionSender.sol   #1

46:                 abi.encode(msg.sender, amount),
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/redeem/RedemptionSender.sol#L46

[G-20] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table

There are 18 instances of this issue.

[G-21] Duplicated `require()` / `revert()` checks should be refactored to a modifier or function

Saves deployment costs

There are 30 instances of this issue.

[G-22] Division by two should use bit shifting

`<x> / 2` is the same as `<x> >> 1`. The `DIV` opcode costs **5 gas**, whereas `SHR` only costs **3 gas**

There are 9 instances of this issue.

[G-23] Stack variable used as a cheaper cache for a state variable is only used once

If the variable is only accessed once, it's cheaper to use the state variable directly that one time

There is 1 instance of this issue:

```
File: contracts/contracts/VotingEscrow.sol   #1
```

```
1047:         uint _epoch = epoch;
```

https://github.com/code-423n4/2022-05-velodrome/blob/7fda97c570b758bbfa7dd6724a336c43d4041740/contracts/contracts/VotingEscrow.sol#L1047

[G-24] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables

There are 13 instances of this issue.

[G-25] Use custom errors rather than `revert()` / `require()` strings to save deployment gas

Custom errors are available from solidity version 0.8.4. The instances below match or exceed that version

There are 86 instances of this issue.

Alex the Entreprenerd (judge) commented:

> The warden could have sent the 5 immutable variables findings and could have probably won.

> Just the 5 immutable variables would save 2100 * 5 = 10500 gas for end users by saving at least on SLOAD.

> Will rate the rest of the savings after reviewing other reports.

Alex the Entreprenerd (judge) commented:

> Very thorough report, that I think missed 1 immutable variable. Overall great, however some findings would benefit by not pasting the same “mistake” 30+ times.

> Total Gas Saved: 17931

> Winning report, well played!

> *(Note: See* **original submission** *for judge's full commentary.)*

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top