Velodrome Finance Security Review

Auditors

0xLeastwood, Lead Security Researcher

Jonah1005, Lead Security Researcher

Alex the Entreprenerd, Security Researcher

Xiaoming90, Security Researcher

Jonatas Martins, Junior Security Researcher

0xNazgul, Junior Security Researcher

Report prepared by: Pablo Misirov

July 17, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Velodrome is next generation DEX that offers low-slippages token swaps, capital-efficient liquidity, and sustainable yields. At it's core, is a solution for protocols on Optimism to properly incentivize liquidity for their own use cases.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Velodrome according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Velodrome Finance engaged with Spearbit to review the Velodrome Finance V2 protocol. In this period of time a total of **119** issues were found.

Note: Post-engagement review information can be found in the **Appendix** section at the bottom of this document.

After the review, all instances of 'contracts-private' located in the context field of this document have been changed to 'contracts' as per client request.

Summary

Project Name	Velodrome Finance
Repository	contracts
Commit	80dd33...bcff
Type of Project	DEX, DeFi
Audit Timeline	Feb 7 - Feb 20
Two week fix period	Feb 21 - Mar 7
Post-engagement	May 22 - May 26, June 12 - June 16

Total Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	8	8	0
Medium Risk	19	16	3
Low Risk	30	18	12
Gas Optimizations	16	13	3
Informational	45	24	21
Total	**119**	**80**	**39**

5 Findings

5.1 Critical Risk

5.1.1 `Reward` calculates `earned` incorrectly on each epoch boundary

Severity: Critical Risk

Context: Reward.sol#L161-L192

Description: Rewards are allocated on a per epoch basis to users in proportion to their total deposited amount. Because the balance and total supply used for rewards is based on `_currTs % WEEK + WEEK`, the values will not represent the end of the current epoch, but instead the first second of the next epoch.

As a result, if a user deposits at any epoch boundary, their deposited amount will actually contribute to the checkpointed total supply of the prior epoch. This leads to a few issues which are detailed below:

- Users who deposit in the first second of the next epoch will dilute the total supply for the prior epoch while not being eligible to claim rewards for that same epoch. Consequently, some rewards will be left unclaimed and locked within the contract as the `tokenRewardsPerEpoch` mapping is used to store reward amounts so unclaimed rewards will not roll over to future epochs.
- Users can also avoid zero `numEpochs` by depositing a negligible amount at an earlier epoch for multiple accounts before attempting to deposit a larger amount at `_currTs % WEEK == 0`. The same user can withdraw their deposit from the `VotingEscrow` contract with the claimed rewards and re-deposit these funds into another account in the same block. They are able to abuse this issue to claim all rewards allocated to each epoch.
- In a similar fashion, reward distributions that are weighted by users' votes in the `Voter` contract can suffer the same issue as outlined above. If the attacker votes some negligible amount on various pools using several accounts, they can increase the vote, claim, reset the vote and re-vote via another account to claim rewards multiple times.

The math below shows that `_currTs + WEEK` is indeed the first second of the next epoch and not the last of the prior epoch.

```
 uint256 internal constant WEEK = 7 days;
 function epochStart(uint256 timestamp) internal pure returns (uint256) {
   return timestamp - (timestamp % WEEK);
}

 epochStart(123)
Type: uint
 Hex: 0x0
 Decimal: 0
 epochStart(100000000)
Type: uint
 Hex: 0x5f2b480
 Decimal: 99792000
 WEEK
Type: uint
 Hex: 0x93a80
 Decimal: 604800
 epochStart(WEEK)
Type: uint
 Hex: 0x93a80
 Decimal: 604800
 epochStart(1 + WEEK)
Type: uint
 Hex: 0x93a80
 Decimal: 604800
 epochStart(0 + WEEK)
Type: uint
 Hex: 0x93a80
 Decimal: 604800
 epochStart(WEEK - 1)
Type: uint
 Hex: 0x0
 Decimal: 0
```

Recommendation: Both lines which query the user's prior balance and total supply for each given epoch can be replaced to check only the last second of the epoch:

```
 if (numEpochs > 0) {
    for (uint256 i = 0; i < numEpochs; i++) {
        // get index of last checkpoint in this epoch
+       _index = getPriorBalanceIndex(tokenId, _currTs + DURATION - 1);
-       _index = getPriorBalanceIndex(tokenId, _currTs + DURATION);
        // get checkpoint in this epoch
        cp0 = checkpoints[tokenId][_index];
        // get supply of last checkpoint in this epoch
+       _supply = Math.max(supplyCheckpoints[getPriorSupplyIndex(_currTs + DURATION - 1)].supply, 1);
-       _supply = Math.max(supplyCheckpoints[getPriorSupplyIndex(_currTs + DURATION)].supply, 1);
        reward += (cp0.balanceOf * tokenRewardsPerEpoch[token][_currTs]) / _supply;
        _currTs += DURATION;
    }
 }
```

This will ensure that users who deposit at the start of the following epoch will not be eligible for rewards or dilute the supply when `block.timestamp % WEEK == 0`.

Velodrome: Fixed in commit 02e0bc.

Spearbit: Verified.

5.2 High Risk

5.2.1 DOS attack by delegating tokens at `MAX_DELEGATES = 1024`

Severity: High Risk

Context: VotingEscrow.sol#L1212

Description: Any user can delegate the balance of the locked NFT amount to anyone by calling `delegate`. As the delegated tokens are maintained in an array that's vulnerable to DOS attack, the `VotingEscrow`has a safety check of `MAX_DELEGATES = 1024` preventing an address from having a huge array. Given the current implementation, any user with 1024 delegated tokens takes approximately 23M gas to `transfer/burn/mint` a token. However, the current gas limit of the op chain is 15M. (ref: Op-scan)

- The current `votingEscrow` has a limit of `MAX_DELEGATES=1024`. it's approx 23M to transfer/withdraw a token when there are 1024 delegated voting on a token.
- It's cheaper to delegate from an address with a shorter token list to an address with a longer token list. => If someone trying to attack a victim's address by creating a new address, a new lock, and delegating to the victim. By the time the attacker hit the gas limit, the victim can not withdraw/transfer/delegate.

Recommendation: There's currently no clear hard limit of block size in OP's spec. There's also a chance the OP's sequencer will include a jumbo tx if funds get locked because of out of gas. Nevertheless, there's no precedent example of such cases and it's not a desirable situation for users to deal with the risks. Hence recommend to take to:

1. Adjust the `MAX_DELEGATES=1024`to `128`;
2. Give an option for users to opt-out/opt-in. Users will only accept the delegated tokens if they `opt-in`; or users can opt-out to refuse any uncommissioned delegated tokens.

Also, recommend adding the following test in `VotingEscrow.t.sol`

```
contract VotingEscrowTest is BaseTest {
    function testDelegateLimitAttack() public {
        vm.prank(address(owner));
        VELO.transfer(address(this), TOKEN_1M);
        VELO.approve(address(escrow), type(uint256).max);
        uint tokenId = escrow.createLock(TOKEN_1, 7 days);
        for(uint256 i = 0; i < escrow.MAX_DELEGATES() - 1; i++) {
            vm.roll(block.number + 1);
            vm.warp(block.timestamp + 2);
            address fakeAccount = address(uint160(420 + i));
            VELO.transfer(fakeAccount, 1 ether);
            vm.startPrank(fakeAccount);
            VELO.approve(address(escrow), type(uint256).max);
            escrow.createLock(1 ether, MAXTIME);
            escrow.delegate(address(this));
            vm.stopPrank();
        }
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + 7 days);
        uint initialGas = gasleft();
        escrow.withdraw(tokenId);
        uint gasUsed = initialGas - gasleft();
        // @audit: setting 10_000_000 to demonstrate the issue. 2~3M gas limit would be a safer range.
        assertLt(gasUsed, 10_000_000);
    }
}
```

Velodrome: Fixed in commit 0b47fe. Delegation was reworked to use static balances so that it would no longer have a limit. This required the introduction of permanent locks which are locks that do not decay.

Spearbit: Verified

5.2.2 Inflated voting balance due to duplicated `veNFTs` within a checkpoint

Severity: High Risk

Context: VotingEscrow.sol#L1309, VotingEscrow.sol#L1364

Description:

> Note: This issue affects `VotingEscrow._moveTokenDelegates` and `VotingEscrow._moveAllDelegates` functions

A checkpoint can contain duplicated veNFTs (tokenIDs) under certain circumstances leading to double counting of voting balance. Malicious users could exploit this vulnerability to inflate the voting balance of their accounts and participate in governance and gauge weight voting, potentially causing loss of assets or rewards for other users if the inflated voting balance is used in a malicious manner (e.g. redirect rewards to gauges where attackers have a vested interest).

Following is the high-level pseudo-code of the existing `_moveTokenDelegates` function, which is crucial for understanding the issue.

1. Assuming moving tokenID=888 from Alice to Bob.
2. Source Code Logic (Moving tokenID=888 out of Alice)
   - Fetch the existing Alice's token IDs and assign them to `srcRepOld`
   - Create a new empty array = `srcRepNew`
   - Copy all the token IDs in `srcRepOld` to `srcRepNew` except for tokenID=888
3. Destination Code Logic (Moving tokenID=888 into Bob)
   - Fetch the existing Bobs' token IDs and assign them to `dstRepOld`
   - Create a new empty array = `dstRepNew`
   - Copy all the token IDs in `dstRepOld` to `dstRepNew`
   - Copy tokenID=888 to `dstRepNew`

The existing logic works fine as long as a new empty array (srcRepNew OR dstRepNew) is created every single time. The code relies on the `_findWhatCheckpointToWrite` function to return the index of a new checkpoint.

```
function _moveTokenDelegates(
..SNIP..
uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
uint256[] storage srcRepNew = _checkpoints[srcRep][nextSrcRepNum].tokenIds;
```

However, the problem is that the `_findWhatCheckpointToWrite` function does not always return the index of a new checkpoint (Refer to Line 1357 below). It will return the last checkpoint if it has already been written once within the same block.

```
function _findWhatCheckpointToWrite(address account) internal view returns (uint32) {
    uint256 _blockNumber = block.number;
    uint32 _nCheckPoints = numCheckpoints[account];

    if (_nCheckPoints > 0 && _checkpoints[account][_nCheckPoints - 1].fromBlock == _blockNumber) {
        return _nCheckPoints - 1;
    } else {
        return _nCheckPoints;
    }
}
```

If someone triggers the `_moveTokenDelegates` more than once within the same block (e.g. perform NFT transfer twice to the same person), the `_findWhatCheckpointToWrite` function will return a new checkpoint in the first transfer but will return the last/previous checkpoint in the second transfer. This will cause the move token delegate logic to be off during the second transfer.

First Transfer at Block 1000

Assume the following states:

```
numCheckpoints[Alice] = 1
_checkpoints[Alice][0].tokenIds = [n1, n2] <== Most recent checkpoint

numCheckpoints[Bob] = 1
_checkpoints[Bob][0].tokenIds = [n3] <== Most recent checkpoint
```

To move tokenID=2 from Alice to Bob, the `_moveTokenDelegates(Alice, Bob, n2)` function will be triggered.

The `_findWhatCheckpointToWrite` will return the index of `1` which points to a new array.

The end states of the first transfer will be as follows:

```
numCheckpoints[Alice] = 2
_checkpoints[Alice][0].tokenIds = [n1, n2]
_checkpoints[Alice][1].tokenIds = [n1] <== Most recent checkpoint

numCheckpoints[Bob] = 2
_checkpoints[Bob][0].tokenIds = [n3]
_checkpoints[Bob][1].tokenIds = [n2, n3] <== Most recent checkpoint
```

Everything is working fine at this point in time.

Second Transfer at Block 1000 (same block)

To move tokenID=1 from Alice to Bob, the `_moveTokenDelegates(Alice, Bob, n1)` function will be triggered.

This time round since the last checkpoint block is the same as the current block, the `_findWhatCheckpointToWrite` function will return the last checkpoint instead of a new checkpoint.

The `srcRepNew` and `dstRepNew` will end up referencing the old checkpoint instead of a new checkpoint. As such, the `srcRepNew` and `dstRepNew` array will reference back to the old checkpoint `_checkpoints[Alice][1].tokenIds` and `_checkpoints[Bob][1].tokenIds` respectively.

The end state of the second transfer will be as follows:

```
numCheckpoints[Alice] = 3
_checkpoints[Alice][0].tokenIds = [n1, n2]
_checkpoints[Alice][1].tokenIds = [n1] <== Most recent checkpoint

numCheckpoints[Bob] = 3
_checkpoints[Bob][0].tokenIds = [n3]
_checkpoints[Bob][1].tokenIds = [n2, n3, n2, n3, n1] <== Most recent checkpoint
```

Four (4) problems could be observed from the end state:

1. The `numCheckpoints` is incorrect. Should be two (2) instead to three (3)
2. TokenID=1 has been added to Bob's Checkpoint, but it has not been removed from Alice's Checkpoint
3. Bob's Checkpoint contains duplicated tokenIDs (e.g. there are two TokenID=2 and TokenID=3)
4. TokenID is not unique (e.g. TokenID appears more than once)

Since the token IDs within the checkpoint will be used to determine the voting power, the voting power will be inflated in this case as there will be a double count of certain NFTs.

```
function _moveTokenDelegates(
..SNIP..
    uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
    uint256[] storage srcRepNew = _checkpoints[srcRep][nextSrcRepNum].tokenIds;
```

Additional Comment about `nextSrcRepNum` variable and `_findWhatCheckpointToWrite` function

In Line 1320 below, the code wrongly assumes that the `_findWhatCheckpointToWrite` function will always return the index of the next new checkpoint. The `_findWhatCheckpointToWrite` function will return the index of the latest checkpoint instead of a new one if `block.number == checkpoint.fromBlock`.

```
function _moveTokenDelegates(
    address srcRep,
    address dstRep,
    uint256 _tokenId
    ) internal {
        if (srcRep != dstRep && _tokenId > 0) {
            if (srcRep != address(0)) {
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256[] storage srcRepOld = srcRepNum > 0
                    ? _checkpoints[srcRep][srcRepNum - 1].tokenIds
                    : _checkpoints[srcRep][0].tokenIds;
                uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
                uint256[] storage srcRepNew = _checkpoints[srcRep][nextSrcRepNum].tokenIds;
```

Additional Comment about `numCheckpoints`

In Line 1330 below, the function computes the new number of checkpoints by incrementing the `srcRepNum` by one. However, this is incorrect because if `block.number == checkpoint.fromBlock`, then the number of checkpoints remains the same and does not increment.

```
function _moveTokenDelegates(
    address srcRep,
    address dstRep,
    uint256 _tokenId
) internal {
    if (srcRep != dstRep && _tokenId > 0) {
        if (srcRep != address(0)) {
            uint32 srcRepNum = numCheckpoints[srcRep];
            uint256[] storage srcRepOld = srcRepNum > 0
                ? _checkpoints[srcRep][srcRepNum - 1].tokenIds
                : _checkpoints[srcRep][0].tokenIds;
            uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
            uint256[] storage srcRepNew = _checkpoints[srcRep][nextSrcRepNum].tokenIds;
            // All the same except _tokenId
            for (uint256 i = 0; i < srcRepOld.length; i++) {
                uint256 tId = srcRepOld[i];
                if (tId != _tokenId) {
                    srcRepNew.push(tId);
                }
            }

            numCheckpoints[srcRep] = srcRepNum + 1;
        }
```

Recommendation: Update the move token delegate logic within the affected functions (`VotingEscrow._moveTokenDelegates` and `VotingEscrow._moveAllDelegates`) to ensure that the latest checkpoint is overwritten correctly when the functions are triggered more than once within a single block.

Further, ensure that the following invariants hold in the new code:

- No duplicated veNFTs (tokenIDs) within a checkpoint
- When moving a tokenID, it must be deleted from the source `tokenIds` list and added to the destination `tokenIds` list
- No more than one checkpoint within the same block for an account. Otherwise, the binary search within the `VotingEscrow.getPastVotesIndex` will return an incorrect number of votes

Sidenote: Another separate issue is that the `fromBlock` of a checkpoint is not set anywhere in the codebase. Therefore, the `_findWhatCheckpointToWrite` function will always create and return a new checkpoint, which is not working as intended. This issue will be raised in another report *"The fromBlock variable of a checkpoint is not initialized"*. Since the remediation of this issue also depends on fixing the `fromBlock` problem, this is being highlighted here again for visibility.

Velodrome: Fixed in commit a670bf.

Spearbit: Verified.

5.2.3 Rebase rewards cannot be claimed after a `veNFT` expires

Severity: High Risk

Context: RewardsDistributor.sol#L271, RewardsDistributor.sol#L283

Description:

> Note: This issue affects both the `RewardsDistributor.claim` and `RewardsDistributor.claimMany` functions

A user will claim their rebase rewards via the `RewardsDistributor.claim` function, which will trigger the `VotingEscrow.deposit_for` function.

```
function claim(uint256 _tokenId) external returns (uint256) {
    if (block.timestamp >= timeCursor) _checkpointTotalSupply();
    uint256 _lastTokenTime = lastTokenTime;
    _lastTokenTime = (_lastTokenTime / WEEK) * WEEK;
    uint256 amount = _claim(_tokenId, _lastTokenTime);
if (amount != 0) {
        IVotingEscrow(ve).depositFor(_tokenId, amount);
    tokenLastBalance -= amount;
    }
    return amount;
}
```

Within the `VotingEscrow.deposit_for` function, the `require` statement at line 812 below will verify that the veNFT performing the claim has not expired yet.

```
function depositFor(uint256 _tokenId, uint256 _value) external nonReentrant {
    LockedBalance memory oldLocked = _locked[_tokenId];

    require(_value > 0, "VotingEscrow: zero amount");
    require(oldLocked.amount > 0, "VotingEscrow: no existing lock found");
    require(oldLocked.end > block.timestamp, "VotingEscrow: cannot add to expired lock, withdraw");
    _depositFor(_tokenId, _value, 0, oldLocked, DepositType.DEPOSIT_FOR_TYPE);
}
```

If a user claims the rebase rewards after their veNFT's lock has expired, the `VotingEscrow.depositFor` function will always revert. As a result, the accumulated rebase rewards will be stuck in the `RewardsDistributor` contract and users will not be able to retrieve them.

Recommendation: Consider sending the claimed VELO rewards to the owner of the veNFT if the veNFT's lock has already expired.

```
 function claim(uint256 _tokenId) external returns (uint256) {
    if (block.timestamp >= timeCursor) _checkpointTotalSupply();
        uint256 _lastTokenTime = lastTokenTime;
        _lastTokenTime = (_lastTokenTime / WEEK) * WEEK;
        uint256 amount = _claim(_tokenId, _lastTokenTime);
    if (amount != 0) {
-       IVotingEscrow(ve).depositFor(_tokenId, amount);
+       IVotingEscrow.LockedBalance memory _locked = IVotingEscrow(ve).locked(_tokenId)
+       if (_locked.end < block.timestamp) {
+           address _nftOwner = IVotingEscrow(ve).ownerOf(_tokenId);
+           IERC20(token).transfer(_nftOwner, amount);
+       } else {
+           IVotingEscrow(ve).depositFor(_tokenId, amount);
+       }
        tokenLastBalance -= amount;
    }
    return amount;
 }
```

Velodrome: Fixed in commit 8a71a8.

Spearbit: Verified.

5.2.4 Claimed rebase rewards of managed NFT are not compounded within `LockedManagedReward`

Severity: High Risk

Context: VotingEscrow.sol#L165, RewardsDistributor.sol#L271

Description: Rebase rewards of a managed NFT should be compounded within the `LockedManagedRewards` contract. However, this was not currently implemented.

When someone calls the `RewardsDistributor.claim` with a managed NFT, the claimed rebase rewards will be locked via the `VotingEscrow.depositFor` function (Refer to Line 277 below). However, the `VotingEscrow.depositFor` function fails to notify the `LockedManagedRewards` contract of the incoming rewards. Thus, the rewards do not accrue in the `LockedManagedRewards`.

```
function claim(uint256 _tokenId) external returns (uint256) {
    if (block.timestamp >= timeCursor) _checkpointTotalSupply();
    uint256 _lastTokenTime = lastTokenTime;
    _lastTokenTime = (_lastTokenTime / WEEK) * WEEK;
    uint256 amount = _claim(_tokenId, _lastTokenTime);
    if (amount != 0) {
        IVotingEscrow(ve).depositFor(_tokenId, amount);
        tokenLastBalance -= amount;
    }
    return amount;
}
```

One of the purposes of the `LockedManagedRewards` contract is to accrue rebase rewards claimed by the managed NFT so that the users will receive their pro-rata portion of the rebase rewards based on their contribution to the managed NFT when they withdraw their normal NFTs from the managed NFT via the `VotingEscrow.withdrawManaged` function.

```
/// @inheritdoc IVotingEscrow
function withdrawManaged(uint256 _tokenId) external nonReentrant {
..SNIP..
    uint256 _reward = IReward(_lockedManagedReward).earned(address(token), _tokenId);
..SNIP..
// claim locked rewards (rebases + compounded reward)
    address[] memory rewards = new address[](1);
    rewards[0] = address(token);
    IReward(_lockedManagedReward).getReward(_tokenId, rewards);
```

If the rebase rewards are not accrued in the `LockedManagedRewards`, users will not receive their pro-rata portion of the rebase rewards during withdrawal.

Recommendation: Ensure that rebase rewards claimed by a managed NFT are accrued to `LockedManagedRewards` contract.

The `depositFor` function could be modified so that any deposits to a managed NFT will notify the `LockedManagedRewards` contract

Velodrome: Acknowledged and will fix. Fixed in commit 632c36 and commit e98472.

Spearbit: Observed the following fixes in commit 632c36:

- Added validation to ensure that the `depositFor` function cannot be called against a Locked NFT
- If `depositFor` function is called against a Managed NFT, the deposited amount will be treated as locked rewards and it will notify the `LockedManagedReward` contract

Observed the following fixes in commit e98472:

- `depositFor` function cannot be called with a locked NFT
- Only `RewardDistributor` can call `depositFor` function with a managed NFT
- If `depositFor` function is called with a managed NFT, the function will notify the `LockedManagedReward` contract

Based on the above, it was observed that the claimed rebase rewards of managed NFT are compounded within the `LockedManagedReward` contract. Verified.

5.2.5 Malicious users could deposit normal NFTs to a managed NFT on behalf of others without their permission

Severity: High Risk

Context: VotingEscrow.sol#L130

Description: The `VotingEscrow.depositManaged` function did not verify that the caller (`msg.sender`) is the owner of the `_tokenId`. As a result, a malicious user can deposit normal NFTs to a managed NFT on behalf of others without their permission.

```
function depositManaged(uint256 _tokenId, uint256 _mTokenId) external nonReentrant {
    require(escrowType[_mTokenId] == EscrowType.MANAGED, "VotingEscrow: can only deposit into managed
    ↪  nft");
    require(!deactivated[_mTokenId], "VotingEscrow: inactive managed nft");
    require(escrowType[_tokenId] == EscrowType.NORMAL, "VotingEscrow: can only deposit normal nft");
    require(!voted[_tokenId], "VotingEscrow: nft voted");
    require(ownershipChange[_tokenId] != block.number, "VotingEscrow: flash nft protection");
    require(_balanceOfNFT(_tokenId, block.timestamp) > 0, "VotingEscrow: no balance to deposit");
..SNIP..
```

The owner of a normal NFT will have their voting balance transferred to a malicious managed NFT, resulting in loss of rewards and voting power for the victim. Additionally, a malicious owner of a managed NFT could aggregate

voting power of the victim's normal NFTs, and perform malicious actions such as stealing the rewards from the victims or use its inflated voting power to pass malicious proposals.

Recommendation: Implement additional validation to ensure that the caller is the owner of the deposited NFT.

```
 function depositManaged(uint256 _tokenId, uint256 _mTokenId) external nonReentrant {
+   address sender = _msgSender();
+   require(_isApprovedOrOwner(sender, _tokenId), "VotingEscrow: not owner or approved");
        require(escrowType[_mTokenId] == EscrowType.MANAGED, "VotingEscrow: can only deposit into
        ↪  managed nft");
        require(!deactivated[_mTokenId], "VotingEscrow: inactive managed nft");
..SNIP..
```

Velodrome: Acknowledged and will fix. Fixed in commit 712e14.

Spearbit: Verified.

5.2.6 First liquidity provider of a stable pair can DOS the pool

Severity: High Risk

Context: Pair.sol#L504, Pair.sol#L353

Description: The invariant k of a stable pool is calculated as follows Pair.sol#L505

```
function _k(uint256 x, uint256 y) internal view returns (uint256) {
    if (stable) {
        uint256 _x = (x * 1e18) / decimals0;
        uint256 _y = (y * 1e18) / decimals1;
        uint256 _a = (_x * _y) / 1e18;
        uint256 _b = ((_x * _x) / 1e18 + (_y * _y) / 1e18);
        return (_a * _b) / 1e18; // x3y+y3x >= k
    } else {
        return x * y; // xy >= k
    }
}
```

The value of `_a = (x * y ) / 1e18` = 0 due to rounding error when `x*y < 1e18`. The rounding error can lead to the invariant k of stable pools equals zero, and the trader can steal whatever is left in the pool.

The first liquidity provider can DOS the pair by: 1.`mint` a small amount of liquidity to the pool, 2. Steal whatever is left in the pool, 3. Repeat step 1, and step 2 until the overflow of the total supply.

To prevent the issue of rounding error, the reserve of a pool should never go too small. The mint function which was borrowed from uniswapV2 has a minimum liquidity check of `sqrt(a * b) > MINIMUM_LIQUIDITY`; This, however, isn't safe enough to protect the invariant formula of stable pools. Pair.sol#L344-L363

```
uint256 internal constant MINIMUM_LIQUIDITY = 10**3;
// ...
function mint(address to) external nonReentrant returns (uint256 liquidity) {
    // ...
    uint256 _amount0 = _balance0 - _reserve0;
    uint256 _amount1 = _balance1 - _reserve1;
    uint256 _totalSupply = totalSupply();
    if (_totalSupply == 0) {
        liquidity = Math.sqrt(_amount0 * _amount1) - MINIMUM_LIQUIDITY;
        //@audit what about the fee?
        _mint(address(1), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens -
        ↪  cannot be address(0)
// ...
}
```

This is the POC of an exploit extended from Pair.t.sol

```
contract PairTest is BaseTest {
// ...
    function drainPair(Pair pair, uint initialFraxAmount, uint initialDaiAmount) internal {
        DAI.transfer(address(pair), 1);
        uint amount0;
        uint amount1;

        if (address(DAI) < address(FRAX)) {
            amount0 = 0;
            amount1 = initialFraxAmount - 1;
        } else {
            amount1 = 0;
            amount0 = initialFraxAmount - 1;
        }

        pair.swap(amount0, amount1, address(this), new bytes(0));
        FRAX.transfer(address(pair), 1);
        if (address(DAI) < address(FRAX)) {
            amount0 = initialDaiAmount; // initialDaiAmount + 1 - 1
            amount1 = 0;
        } else {
            amount1 = initialDaiAmount; // initialDaiAmount + 1 - 1
            amount0 = 0;
        }
        pair.swap(amount0, amount1, address(this), new bytes(0));
    }

    function testDestroyPair() public {
        deployCoins();
        deal(address(DAI), address(this), 100 ether);
        deal(address(FRAX), address(this), 100 ether);

        deployFactories();
        Pair pair = Pair(factory.createPair(address(DAI), address(FRAX), true));
        for(uint i = 0; i < 10; i++) {
            DAI.transfer(address(pair), 10_000_000);
            FRAX.transfer(address(pair), 10_000_000);
            // as long as 10_000_000^2 < 1e18
            uint liquidity = pair.mint(address(this));
            console.log("pair:", address(pair), "liquidity:", liquidity);
            console.log("total liq:", pair.balanceOf(address(this)));
            drainPair(pair, FRAX.balanceOf(address(pair)) , DAI.balanceOf(address(pair)));
            console.log("DAI balance:", DAI.balanceOf(address(pair)));
            console.log("FRAX balance:", FRAX.balanceOf(address(pair)));
            require(DAI.balanceOf(address(pair)) == 1, "should drain DAI balance");
            require(FRAX.balanceOf(address(pair)) == 2, "should drain FRAX balance");
        }
        DAI.transfer(address(pair), 1 ether);
        FRAX.transfer(address(pair), 1 ether);
        vm.expectRevert();
        pair.mint(address(this));
    }
}
```

Recommendation: Recommend to add two restrictions on the first lp of stable pools:

1. only allows equal amounts of liquidity.
2. invariant `_k` should be larger than the MINIMUM_K

```
    function mint(address to) external nonReentrant returns (uint256 liquidity) {
   // ...
        if (_totalSupply == 0) {
            liquidity = Math.sqrt(_amount0 * _amount1) - MINIMUM_LIQUIDITY;
            _mint(address(1), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY
            ↪  tokens - cannot be address(0)
+            if (stable) {
+               require(_amount0 * 1e18 / decimals0 == _amount1 * 1e18 / decimals1,  "Pair: Stable pair
↪  must be equal amounts");
+                require(_k(_amount0, _amount1) > MINIMUM_K, "Pair: Stable pair must be above minimum
↪  k");
+            }
        }
    // ...
 }
```

Velodrome: I agree with this finding. `MINIMUM_K` is not specified, but I assume it to be equal to zero. We will fix it.

Is there any reason why the initial deposit must be equal? In practice, stable pools often aren't perfectly 1:1 pegged, and if the difference is significant, this change will guarantee that the initial deposit will always be small. It may not make much difference but just want to know all the thinking behind it.

Spearbit: Agreed that enforcing the 1:1 ratio forces first lp lose money in many cases. We can prob remove it if setting a higher `MINIMUM_K`.

The reason to enforce the 1:1 ratio is just a precautions check. It's when lp can contribute to max invariant k with minimum tokens; the issue of rounding error should be mildest.

As for `MINIMUM_K`, `10e9` would be a safe one. The invariant can decrease in some cases.

The `x3y+y3x` AMM is quite unique. This is the less-studied curve among all popular AMM. We will revisit this issue and "the decreased invariant one" and figure out a tighter safe bound after going through the other part of the protocol. At the time, I tend to keep the security margin and set a higher `MINIMUM_K`.

Velodrome: Fixed in commit 59f9c1.

Spearbit: Verified.

5.3 Medium Risk

5.3.1 Certain functions are unavailable after opting in to the "auto compounder

Severity: Medium Risk

Context: AutoCompounder.sol

Description: Certain features (e.g., delegation, governance voting) of a veNFT would not be available if the veNFT is transferred to the auto compounder.

Let *x* be a managed veNFT. When an "auto compounder" is created, ownership of *x* is transferred to the `AutoCompounder` contract, and any delegation within *x* is cleared.

The auto compounder can perform gauge weight voting using *x* via the provided `AutoCompounder.vote` function. However, it loses the ability to perform any delegation with *x* because the `AutoCompounder` contract does not contain a function that calls the `VotingEscrow.delegate` function. Only the owner of *x*, the `AutoCompounder` contract, can call the `VotingEscrow.delegate` function.

x also loses the ability to vote on governance proposals as the existing `AutoCompounder` contract does not support this feature.

Once the owner of the managed NFTs has opted in to the "auto compounder," it is not possible for them to subsequently "opt out." Consequently, if they need to exercise delegation and governance voting, they will be unable to do so, exacerbating the impact.

Recommendation: Most users would expect their managed NFTs to function similarly after moving them to the "auto compounder" with the extra benefit of automatically compounding the rewards. Thus, it is recommended that any features available to a managed NFT should also be ported over to the "auto compounder".

Otherwise, document these limitations so that the users would be aware of this before using the "auto compounder"

Velodrome: The design to prevent voting/delegation from the managed veNFT ((m)veNFT) is intentional. There is potential for Velodrome governance abuse by an (m)veNFT that gains too much voting power from its' underlying locked veNFTs. We completed similar research for LP delegation and determined it is better for the protocol to prevent overweight voting from entities who have voting power not intentionally given to them. In this case, a user would be locking up their veNFT with the intention of receiving the auto-compounding returns, not delegating their voting rights away.

Spearbit: Acknowledged.

5.3.2 Claimable gauge distributions are locked when `killGauge` is called

Severity: Medium Risk

Context: Voter.sol#L297-L311

Description: When a gauge is killed, the `claimable[_gauge]` key value is cleared. Because any rewards received by the `Voter` contract are indexed and distributed in proportion to each pool's weight, this claimable amount is permanently locked within the contract.

Recommendation: Consider returning the `claimable` amount to the `Minter` contract. It is important to note that votes will continue to persist on the killed gauge, so it may also make sense to wipe these votes too. Otherwise, the killed gauge will continue to accrue rewards from the `Minter` contract.

Velodrome: We intend on returning claimable to Minter. I think clearing votes is not possible without a lot of changes to the code as there is no way of fetching which nfts voted for a specific pool without iterating through all of them and this may have unexpected side effects with `reset`, so I think the best that can be done is to communicate that the gauge has been killed.

Velodrome: Fixed in commit e4b230.

Spearbit: The fix will send a gauge's claimable funds back to the `Minter` contract in `killGauge()`. However, it does not handle the case where residual votes on a pool will continue to allocate minted tokens to a gauge.

Velodrome: Is it possible for the mitigation to be complete by returning funds in `updateFor(address _gauge)` as well? By complete I mean addressing the edge case that votes remain on the pool, and thus causing minting supply to be trapped.

That way, even if there are residual votes (`_supplied`), any `claimable` that exists is returned to the minter instead of being trapped in `Voter`. Note that `updateFor` is only callable once per week as the `index` value is only updated once per week when `update_period` is called.

Something like this:

```
if (_delta > 0) {
    uint256 _share = (uint256(_supplied) * _delta) / 1e18; // add accrued difference for each supplied
    ↪  token
    if (isAlive[_gauge]) {
        claimable[_gauge] += _share;
    } else {
        IERC20(rewardToken).safeTransfer(minter, _share);
    }
}
```

Spearbit: Agreed that the residual votes issue would be remediated if `updateFor()` was updated to transfer back minted tokens if the gauge has been killed.

5.3.3 Bribe and fee token emissions can be gamed by users

Severity: Medium Risk

Context: Voter.sol#L154-L224

Description: A user may vote or reset their vote once per epoch. Votes persist across epochs and once a user has distributed their votes among their chosen pools, the `poke()` function may be called by any user to update the target user's decayed `veNFT` token balance. However, the `poke()` function is not hooked into any of the reward distribution contracts.

As a result, a user is incentivized to vote as soon as they create their lock and avoid re-voting in subsequent epochs. The amount deposited via `Reward._deposit()` does not decay linearly as how it is defined under veToken mechanics. Therefore, users could continue to earn trading fees and bribes even after their lock has expired. Simultaneously, users can `poke()` other users to lower their voting weight and maximize their own earnings.

Recommendation: Re-designing the `FeesVotingReward` and `BribeVotingRewards` contracts may be worthwhile to decay user deposits automatically.

Velodrome: Given that there are a lot of assumptions around the severity of the attack (i.e. how likely a user would wish to do this, given competing incentives around maximizing returns and the risk that they can be poked at any time, etc), we have elected to go with a solution that involves incentivizing the poking of `tokenId`s that are passively voting for too long.

It will look something like this. On the last day of every epoch, any `tokenId` that is over some value, that has been passively voting for some time, will be incentivized to be poked. This will amend down that NFT's contribution and discourage long-term passive behavior as well as abuse. The parameters ("some value" and "some time") will be mutable and can be modified to reduce the cost of incentivization. Some initial ideas of values would be to poke any NFT worth more than 1_000 VELO that has passively voted for 4 weeks. We will assess the severity of the issue in practice and make changes accordingly.

Spearbit: Acknowledged. While it is true that users are incentivized to `poke` other users who are passively voting on pools for too long, this mechanism is still inefficient and does not feasibly scale with users.

5.3.4 Desync between bribes being paid and gauge distribution allows voters to receive bribes without triggering emissions

Severity: Medium Risk

Context: Voter.sol#L448

Description: Because of the fact that `BribeVotingReward` will award bribes based on the voting power at `EPOCH_END - 1`, but `Minter.update_period()` and `Voter.distribute()` can be called at a time after, some voters may switch their vote before their weight influences emissions, causing the voters to receive bribes, but the bribing protocols to not have their gauge receive the emissions.

For example: Let's say as project XYZ I'm going to bribe for 3 weeks in a row and it will be the best yield offered

What you could do due to the discrepancy between distribute and the rewards is:

- You can vote for 3 weeks
- As soon as the 3rd vote period has ended (Reward epoch), you can vote somewhere else

The 3rd vote will:

- Award you with the 3rd week epoch of rewards
- Can be directed towards a new Gauge, causing the distribution to not be in-line with the Bribes

The desync between:

- Bribes being Paid (`EPOCH_END - 1`) and
- distribution happening some-time after, `EPOCH_END + X`

Means that some bribes can get the reward without "reciprocating" with their vote weight.

Velodrome: Discussed this with the team. We acknowledge that it is an issue, although we do think that economic incentives encourage voters to vote later during the epoch (as they will seek to profit maximize, this was discussed elsewhere in another issue). Due to the potential risk of loss for partners, we considered a simple mitigation of preventing voting in another window (an hour long post epoch flip, similar to the votium-style time window) to allow distributions to take place. Distributions are an automation target and will be incentivized down the track.

I have thought about it and have also considered an option where users passively call distribute prior to voting (e.g. perhaps in reset but it appears it does not mitigate the issue as it requires ALL pools to be distributed (at a fixed total voting weight supply), prior to allowing votes to change.

Thus, I continue to think that the simplest fix is to implement a post-epoch voting window of an hour to allow distributions to take place.

Fixed in commit b517ab.

Spearbit: Verified. Agree that allowing 1 hour post-voting window for protocols to trigger distribute is an appropriate mitigation.

5.3.5 Compromised or malicious owner can drain the `VotingEscrow` contract of `VELO` tokens

Severity: Medium Risk

Context: VotingEscrow.sol#L119-L122, FactoryRegistry.sol#L72-L82

Description: The `FactoryRegistry` contract is an `Ownable` contract with the ability to control the return value of the `managedRewardsFactory()` function. As such, whenever `createManagedLockFor()` is called in `VotingEscrow`, the `FactoryRegistry` contract queries the `managedRewardsFactory()` function and subsequently calls `createRewards()` on this address.

If ownership of the `FactoryRegistry` contract is compromised or malicious, the `createRewards()` external call can return any arbitrary `_lockedManagedReward` address which is then given an infinite token approval. As a result, it's possible for all locked `VELO` tokens to be drained and hence, this poses a significant centralization risk to the protocol.

Recommendation: Avoid using infinite approvals unless the target is guaranteed to be deterministic and immutable. Consider modifying the `increaseAmount()` and all other instances where `IReward(_lockedManagedReward).notifyRewardAmount()` is called.

The proposed fix may look like the following:

```
 function createManagedLockFor(address _to) external nonReentrant returns (uint256 _mTokenId) {
    ...
    (address _lockedManagedReward, address _freeManagedReward) = IManagedRewardsFactory(
        IFactoryRegistry(factoryRegistry).managedRewardsFactory()
    ).createRewards(voter);
-   IERC20(token).approve(_lockedManagedReward, type(uint256).max);
    ...
 }

 function increaseAmount(uint256 _tokenId, uint256 _value) external nonReentrant {
    ...
    if (_escrowType == EscrowType.MANAGED) {
        // increaseAmount called on managed tokens are treated as locked rewards
        address _lockedManagedReward = managedToLocked[_tokenId];
+       IERC20(token).approve(_lockedManagedReward, _value);
        IReward(_lockedManagedReward).notifyRewardAmount(address(token), _value);
    }
 }
```

Velodrome: Fixed in commit 6726f2.

Spearbit: Verified.

5.3.6 Unsafe casting in `RewardsDistributor` leads to underflow of `veForAt`

Severity: Medium Risk

Context: RewardsDistributor.sol#L121-L127

Description: Solidity does not revert when casting a negative number to `uint`. Instead, it underflows to a large number. In the `RewardDistributor` contract, the balance of a token at specific time is calculated as follows

```
IVotingEscrow.Point memory pt = IVotingEscrow(_ve).userPointHistory(_tokenId, epoch);
Math.max(uint256(int256(pt.bias - pt.slope * (int128(int256(_timestamp - pt.ts))))), 0);
```

This supposes to return zero when the calculated balance is a negative number. However, it underflows to a large number.

This would lead to incorrect reward distribution if third-party protocols depend on this function, or when further updates make use of this codebase.

Recommendation: Recommend following other parts of the codebase and returning zero for a negative number.

```
int256 result = int256(pt.bias - pt.slope * int128(int256(_timestamp - pt.ts)));
if (result < 0) return 0;
return uint256(result);
```

Also, recommend applying the fix to other parts of `rewardDistributor` RewardsDistributor.sol#L196 RewardsDistributor.sol#L254 RewardsDistributor.sol#L145

Velodrome: Fixed in commit 6485ef.

Spearbit: Verified.

5.3.7 Proposals can be griefed by front-running and canceling

Severity: Medium Risk

Context: VeloGovernor.sol#L19

Description: Because the Governor uses OZ's Implementation, a griefter can front-run a valid proposal with the same settings and then immediately `cancel` it.

You can avoid the grief by writing a macro contract that generates random descriptions to avoid the front-run.

See: code-423n4/2022-09-nouns-builder-findings#182.

Recommendation: Add `proposer` to the `proposalHash()` function to avoid griefing.

Velodrome: Fixed in commit b2f8f2.

Spearbit: Verified.

5.3.8 `pairFor` does not correctly sort tokens when overriding for `SinkConverter`

Severity: Medium Risk

Context: Router.sol#L59-L90

Description: The router will always search for pairs by sorting tokenA and TokenB.

Notably, for the velo and Velo2 pair, the Router will not perform the sorting

```
//Router.sol#L69-L73
if (factory == defaultFactory) {
     if ((tokenA == IPairFactory(defaultFactory).velo()) && (tokenB ==
     ↪  IPairFactory(defaultFactory).veloV2())) {
         return IPairFactory(defaultFactory).sinkConverter();
    }
}
```

Meaning that the pair for Velo -> Velo2 will be the Sink but the pair for Velo2 -> Velo will be some other pair.

Additionally, you can front-run a call to setSinkConverter() by calling createPair() with the same parameters. However, the respective values for getPair() would be overwritten with the sinkConverter address. This could lead to some weird and unexpected behaviour as we would still have an invalid Pair contract for the v1 and v2 velo tokens.

Recommendation: It may be best to enforce the 1 way direction but sort the pair to ensure that all Velo -> Velo2 go to the Sink.

Velodrome: Fix implemented commit b0adb4. The front-run still exists where a legit `VELO/VELO V2` token pair could be created but from the router perspective, once `PairFactory.setSinkConverter()` is called, the invalid created pair would be ignored.

Spearbit: Verified.

5.3.9 Inconsistent between `balanceOfNFT`, `balanceOfNFTAt` and `_balanceOfNFT` functions

Severity: Medium Risk

Context: VotingEscrow.sol#L985, VotingEscrow.sol#L976

Description: The `balanceOfNFT` function implements a flash-loan protection that returns zero voting balance if `ownershipChange[_tokenId] == block.number`. However, this was not consistently applied to the `balanceOfNFTAt` and `_balanceOfNFT` functions.

```
VotingEscrow.sol
function balanceOfNFT(uint256 _tokenId) external view returns (uint256) {
if (ownershipChange[_tokenId] == block.number) return 0;
return _balanceOfNFT(_tokenId, block.timestamp);
}
```

As a result, Velodrome or external protocols calling the `balanceOfNFT` and `balanceOfNFTAt` external functions will receive different voting balances for the same veNFT depending on which function they called.

Additionally, the internal `_balanceOfNFT` function, which does not have flash-loan protection, is called by the `VotingEscrow.getVotes` function to compute the voting balance of an account. The `VotingEscrow.getVotes` function appears not to be used in any in-scope contracts, however, this function might be utilized by some external protocols or off-chain components to tally the votes. If that is the case, a malicious user could flash-loan the veNFTs to inflate the voting balance of their account.

Recommendation: If the requirement is to have all newly transferred veNFTs (`ownershipChange[_tokenId] == block.number`) have zero voting balance to prevent someone from flash-loaning veNFT to increase their voting balance, the flash-loan protection should be consistently implemented across all the related functions.

Velodrome: I think the current status of this issue is that once timestamp governance is merged in, block-based balance functions will be removed as the contract will adopt timestamps as its official "clock" (see EIP6372). Once it is merged, we will assess the consistency of ownership_change on the various fns.

Spearbit: Acknowledged.

5.3.10 Nudge check will break once limit is reached

Severity: Medium Risk

Context: Minter.sol#L75

Description: Because you're checking both sides, once oldRate reaches the MAX_RATE, every new nudge call will revert.

Meaning that if _newRate ever get's to MAXIMUM_TAIL_RATE or MINIMUM_TAIL_RATE, nudge will stop working.

Recommendation: It's probably best to let the value update and then cap it, without a require.

Any proposal that goes over cap can be allowed to go through, but be forced to be idempotent.

E.g. `newRate  = rate + NUDGE > maxRate ? maxRate` for the upside

And `rate == NUDGE ? 0 : rate - NUDGE` for the decrease.

Velodrome: Fixed in commit 42ee2c. The Velodrome team has modified the downside boundary code to keep it consistent with the upper boundary code. It is noted that the downside boundary is at `1 BPS` as there must always be some emissions.

Spearbit: Mitigation LGTM

- Removed require (prevents reverts)
- If above cap, cap to max
- If below the lower cap, bring back to 1, and avoids the underflow as long as NUDGE is 1, which is fine

Spearbit: Verified.

5.3.11 `ownershipChange` can be sidestepped

Severity: Medium Risk

Context: VotingEscrow.sol#L137-L138

Description: The check there is to prevent adding to managed after a transfer from or creation

```
require(ownershipChange[_tokenId] != block.number, "VotingEscrow: flash nft protection");
```

However, it doesn't prevent adding and removing from other managed tokens, merging, or splitting. For this reason, we can sidestep the lock by splitting

Because `ownershipChange` is updated exclusively on `_transferFrom`, we can side-step it being set by splitting the lock into a new one which will not have the lock.

Recommendation: Consider if the lock is necessary and add additional checks to prevent users from side-stepping. Alternatively, the lock functionality may be removed to focus on maintaining underlying invariants.

Velodrome: Fixed in commit 02e0bc.

Spearbit: Verified.

5.3.12 The `fromBlock` variable of a checkpoint is not initialized

Severity: Medium Risk

Context: VotingEscrow.sol#L1353, VotingEscrow.sol#L1256

Description: A checkpoint contains a `fromBlock` variable which stores the block number the checkpoint is created.

```
/// @notice A checkpoint for marking delegated tokenIds from a given block
struct Checkpoint {
    uint256 fromBlock;
    uint256[] tokenIds;
}
```

However, it was found that the `fromBlock` variable of a checkpoint was not initialized anywhere in the codebase. Therefore, any function that relies on the `fromBlock` of a checkpoint will break.

The `VotingEscrow._findWhatCheckpointToWrite` and `VotingEscrow.getPastVotesIndex` functions were found to rely on the `fromBlock` variable of a checkpoint for computation. The following is a list of functions that calls these two affected functions.

```
_findWhatCheckpointToWrite -> _moveTokenDelegates -> _transferFrom
_findWhatCheckpointToWrite -> _moveTokenDelegates -> _mint
_findWhatCheckpointToWrite -> _moveTokenDelegates -> _burn
_findWhatCheckpointToWrite -> _moveAllDelegates -> _delegate -> delegate/delegateBySig

getPastVotesIndex -> getTokenIdsAt
getPastVotesIndex -> getPastVotes -> GovernorSimpleVotes._getVotes
```

Instance 1 - `VotingEscrow._findWhatCheckpointToWrite` function

The `VotingEscrow._findWhatCheckpointToWrite` function verifies if the `fromBlock` of the latest checkpoint of an account is equal to the current block number. If true, the function will return the index number of the last checkpoint.

```
function _findWhatCheckpointToWrite(address account) internal view returns (uint32) {
    uint256 _blockNumber = block.number;
    uint32 _nCheckPoints = numCheckpoints[account];

    if (_nCheckPoints > 0 && _checkpoints[account][_nCheckPoints - 1].fromBlock == _blockNumber) {
        return _nCheckPoints - 1;
    } else {
        return _nCheckPoints;
    }
}
```

As such, this function does not work as intended and will always return the index of a new checkpoint.

Instance 2 - `VotingEscrow.getPastVotesIndex` function

The `VotingEscrow.getPastVotesIndex` function relies on the `fromBlock` of the latest checkpoint for optimization purposes. If the request block number is the most recently updated checkpoint, it will return the latest index immediately and skip the binary search. Since the `fromBlock` variable is not populated, the optimization will not work.

```
function getPastVotesIndex(address account, uint256 blockNumber) public view returns (uint32) {
    uint32 nCheckpoints = numCheckpoints[account];
    if (nCheckpoints == 0) {
        return 0;
    }
    // First check most recent balance
    if (_checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
        return (nCheckpoints - 1);
    }

    // Next check implicit zero balance
    if (_checkpoints[account][0].fromBlock > blockNumber) {
        return 0;
    }

    uint32 lower = 0;
    uint32 upper = nCheckpoints - 1;
    while (upper > lower) {
        uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
        Checkpoint storage cp = _checkpoints[account][center];
        if (cp.fromBlock == blockNumber) {
            return center;
        } else if (cp.fromBlock < blockNumber) {
            lower = center;
        } else {
            upper = center - 1;
        }
    }
    return lower;
}
```

Recommendation: Initialize the `fromBlock` variable of the checkpoint in the codebase.

Velodrome: Fixed in commit a670bf.

Spearbit: Verified.

5.3.13 Double voting by shifting the voting power between managed and normal NFTs

Severity: Medium Risk

Context: VotingEscrow.sol#L165

Description: Owners of normal NFTs and managed NFTs could potentially collude to double vote, which affects the fairness of the gauge weight voting. A group of malicious veNFT owners could exploit this and use the inflated voting balance to redirect the VELO emission to gauges where they have a vested interest, causing losses to other users.

The following shows that it is possible to increase the weight of a pool by 2000 with a 1000 voting balance within a single epoch by shifting the voting power between managed and normal NFTs.

For simplicity's sake, assume the following

- Alice is the owner of a managed NFT (tokenID=888)
- Bob is the owner of a normal NFT (tokenID=999)
- Alice's managed NFT (tokenID=888) only consists of one (1) normal NFT (tokenID=999) that belongs to Bob being locked up.

The following steps are executed within the same epoch. At the start, the state is as follows:

- Voting power of Alice's managed NFT (tokenID=888) is 1000

  - The 1000 voting came from the normal NFT (tokenID=999) during the deposit
  - `weights[_tokenId][_mTokenId] = _weight | weights[999][888] = 1000;`
- Voting power of Bob's normal NFT (tokenID=999) is zero (0)
- Weight of Pool X = 0

Alice calls `Voter.vote` function with his managed NFT (tokenID=888), and increases the weight of Pool X by 1000. Subsequently, the `lastVoted[_tokenId] = _timestamp` at Line 222 in the `Voter.vote` function will be set, and the `onlyNewEpoch` modifier will ensure Alice cannot use the same managed NFT (tokenID=888) to vote again in the current epoch.

However, Bob could call the `VotingEscrow.withdrawManaged` to withdraw his normal NFT (tokenID=999) from the managed NFT (tokenID=888). Within the function, it will call the internal `_checkpoint` function to "transfer" the voting power from managed NFT (tokenID=888) to normal NFT (tokenID=999).

At this point, the state is as follows:

- Voting power of Alice's managed NFT (tokenID=888) is zero (0)
- Voting power of Bob's normal NFT (tokenID=999) is 1000
- Weight of Pool X = 1000

Bob calls `Voter.vote` function with his normal NFT (tokenID=999) and increases the weight of Pool X by 1000. Since normal NFT (tokenID=999) has not voted in the current epoch, it is allowed to vote. The weight of Pool X becomes 2000.

It was observed that a mechanism is in place to punish and disincentivize malicious behaviors from a managed NFT owner. The protocol's emergency council could deactivate Managed NFTs involved in malicious activities via the `VotingEscrow.setManagedState` function. In addition, the ability to create a managed NFT is restricted to only an authorized manager and protocol's governor. These factors help to mitigate some risks related to this issue to a certain extent.

Recommendation: Consider calling the `Voter.poke` function against the managed NFT (e.g. tokenID=888) automatically within the `VotingEscrow.withdrawManaged` function so that the weight provided to the pools by a managed NFT (e.g. tokenID=888) will be reduced by the voting balance of the normal NFT to be withdrawn.

Velodrome: Fixed in commit 3926ee.

Spearbit: Verified.

5.3.14 `MetaTX` is using the incorrect `Context`

Severity: Medium Risk

Context: Gauge.sol#L12

Description: Throughout the codebase, the code uses `Context` for `_msgSender()`

The implementation chosen will resolve each `_msgSender()` to `msg.sender` which is inconsistent with the goal of allowing `MetaTX`.

Recommendation: Replace the import of `Context` with `ERC2771Context`

Also see: guide-metatx#compile-using-hardhat.

Velodrome: Fixed in commit 84a2d8.

Spearbit: Verified.

5.3.15 `depositFor` function should be restricted to approved NFT types

Severity: Medium Risk

Context: VotingEscrow.sol#L807

Description: The `depositFor` function was found to accept NFT of all types (normal, locked, managed) without restriction.

```
function depositFor(uint256 _tokenId, uint256 _value) external nonReentrant {
    LockedBalance memory oldLocked = _locked[_tokenId];

    require(_value > 0, "VotingEscrow: zero amount");
    require(oldLocked.amount > 0, "VotingEscrow: no existing lock found");
    require(oldLocked.end > block.timestamp, "VotingEscrow: cannot add to expired lock, withdraw");
    _depositFor(_tokenId, _value, 0, oldLocked, DepositType.DEPOSIT_FOR_TYPE);
}
```

Instance 1 - Anyone can call `depositFor` against a locked NFT

Users should not be allowed to increase the voting power of a locked NFT by calling the `depositFor` function as locked NFTs are not supposed to vote. Thus, any increase in the voting balance of locked NFTs will not increase the gauge weight, and as a consequence, the influence and yield of the deposited VELO will be diminished.

In addition, the locked balance will be overwritten when the `veNFT` is later withdrawn from the managed `veNFT`, resulting in a loss of funds.

Instance 2 - Anyone can call `depositFor` against a managed NFT

Only the `RewardsDistributor.claim` function should be allowed to call `depositFor` function against a managed NFT to process rebase rewards claimed and to compound the rewards into the `LockedManagedReward` contract.

However, anyone could also increase the voting power of a managed NFT directly by calling `depositFor` with a `tokenId` of a managed NFT, which breaks the invariant.

Recommendation: Evaluate the type of NFTs (normal, locked, or managed) that can call the `depositFor` function within protocol.

Based on the current design of the protocol:

- Normal NFT - Anyone can call `depositFor` against a normal NFT
- Locked NFT - No one should be able to call `depositFor` against a locked NFT
- Managed NFT - Only `RewardsDistributor.claim` function is allowed to call `depositFor` function against a managed NFT for processing rebase rewards.

Consider implementing the following additional check to disallow anyone from calling `depositFor` function against a Locked NFT:

```
 function depositFor(uint256 _tokenId, uint256 _value) external nonReentrant {
+   EscrowType _escrowType = escrowType[_tokenId];
+   require(_escrowType != EscrowType.LOCKED, "VotingEscrow: Not allowed to call depositFor against
↪   Locked NFT");

    LockedBalance memory oldLocked = _locked[_tokenId];

    require(_value > 0, "VotingEscrow: zero amount");
    require(oldLocked.amount > 0, "VotingEscrow: no existing lock found");
    require(oldLocked.end > block.timestamp, "VotingEscrow: cannot add to expired lock, withdraw");
    _depositFor(_tokenId, _value, 0, oldLocked, DepositType.DEPOSIT_FOR_TYPE);
 }
```

Sidenote: Additionally, the `depositFor` function should be modified to handle incoming Managed NFT's rebase rewards from the `RewardsDistributor`. Refer to the recommendation in *"Claimed rebase rewards of managed NFT are not compounded within LockedManagedReward"*.

Velodrome: Fixed in commit e98472.

Spearbit: Verified.

5.3.16 Lack of vetoer can lead to 51% attack

Severity: Medium Risk

Context: VeloGovernor.sol, EpochGovernor.sol

Description: The veto power is important functionality in a governance system in order to protect from malicious proposals. However there is lack of vetoer in `VeloGovernor` , this might lead to Velodrome losing their veto power unintentionally and open to 51% attack.

With 51% attack a malicous actor can change the governor in `Voter` contract or by pass the tokens whitelist adding new gauge with malicious token.

References

- dialectic.ch/editorial/nouns-governance-attack-2
- code4rena.com/reports/2022-09-nouns-builder/#m-11-loss-of-veto-power-can-lead-to-51-attack

Recommendation: Is recommended to add vetoer with two step validation and add the function to execute a veto. Example of veto function, should adapt the code to use it:

```
function veto(bytes32 _proposalId) external {
  // Ensure the caller is the vetoer
  require(msg.sender == vetoer, "Only vetoer");

    ProposalState status = state(proposalId);

  // Ensure the proposal has not been executed
  require(
      status != ProposalState.Canceled && status != ProposalState.Expired && status !=
      ↪  ProposalState.Executed,
      "Proposal not active"
  );

  // Get the pointer to the proposal
  Proposal storage proposal = proposals[_proposalId];

  // Update the proposal as vetoed
  proposal.vetoed = true;

  emit ProposalVetoed(_proposalId);
}
```

Velodrome: Acknowledged, given the ability of this to severely disrupt normal operation of the protocol. I think the simplest way to implement this would be to use the _cancel function provided in Governor. We will also add the ability to set/change a vetoer, which will most likely be set to emergencyCouncil (will ask for feedback around this). Are there any recommendations regarding parameters that should be set? e.g. the appropriate quorum fraction?

Spearbit: Additional considerations, especially during migration from V1 to V2, there should be a time in which it will be very cheap to attack the Governor without a vetoer, the first attack could be as simple as raising the quorum (which may force the V1 price to raise as it becomes more urgent to migrate it for V2) This may be used for example to enable new tokens which are malicious/privileged with the goal of obtaining emissions from V2 in perpetuity and keeping enough of a headstart to make it impossible for others to catch up Set of attacks:

- Whitelist privileged pair (to steal emissions)
- Raise quorum to make it harder / impossible to catchup
- Set governor to an EOA / Hijack it

- Create a managed lock and refuse to create it for others

Velodrome: Fixed in commit 64fe60.

Spearbit: Verified.

5.4 Low Risk

5.4.1 Compromised or malicious owner can siphon rewards from the `Voter` contract

Severity: Low Risk

Context: FactoryRegistry.sol#L35-L69, Voter.sol#L272-L279

Description: The `createGauge()` function takes a `_gaugeFactory` parameter which is checked to be approved by the `FactoryRegistry` contract. However, the owner of this contract can approve any arbitrary `FactoryRegistry` contract, hence the return value of the `IGaugeFactory(_gaugeFactory).createGauge()` call may return an EOA which steals rewards every time `notifyRewardAmount()` is called.

Recommendation: Consider modifying the `distribute()` function to only approve the available claimable token amount. The `createGauge()` function should also not approve an infinite amount for any potentially untrusted address.

The proposed fix may look like the following:

```
 function createGauge(
    address _pairFactory,
    address _votingRewardsFactory,
    address _gaugeFactory,
    address _pool
 ) external nonReentrant returns (address) {
    ...
    require(
        IFactoryRegistry(factoryRegistry).isApproved(_pairFactory, _votingRewardsFactory,
        ↪  _gaugeFactory),
        "Voter: factory path not approved"
    );
    (address _feeVotingReward, address _bribeVotingReward) =
    ↪  IVotingRewardsFactory(_votingRewardsFactory)
        .createRewards(rewards);

    address _gauge = IGaugeFactory(_gaugeFactory).createGauge(_pool, _feeVotingReward, rewardToken,
    ↪  isPair);

-       IERC20(rewardToken).approve(_gauge, type(uint256).max);
    ...
}

 function distribute(address _gauge) public nonReentrant {
    IMinter(minter).update_period();
    _updateFor(_gauge); // should set claimable to 0 if killed
    uint256 _claimable = claimable[_gauge];
    if (_claimable > IGauge(_gauge).left() && _claimable / DURATION > 0) {
        claimable[_gauge] = 0;
+           IERC20(rewardToken).approve(_gauge, _claimable);
        IGauge(_gauge).notifyRewardAmount(_claimable);
        emit DistributeReward(_msgSender(), _gauge, _claimable);
    }
 }
```

Velodrome: Fixed in commit 6726f2.

Spearbit: Verified.

5.4.2 Missing `nonReentrant` modifier on a state changing `checkpoint` function

Severity: Low Risk

Context: VotingEscrow.sol#L802-L804

Description: The `checkpoint()` function will call the internal `_checkpoint()` function which ultimately fills the point history and potentially updates the `epoch` state variable.

Recommendation: Add the `nonReentrant` modifier to the external `checkpoint()` function.

Velodrome: Fixed in commit 9317ea.

Spearbit: Verified.

5.4.3 Close to half of the trading fees may be paid one epoch late

Severity: Low Risk

Context: Gauge.sol#L71

Description: Due to how `left()` is implemented in `Reward` (returning the total amount of rewards for the epoch), `_claimFees()` will not queue rewards until the new fees are greater than the current ones for the epoch.

This can cause the check to be false for values that are up to half minus one reward. Consider the following example:

- First second of a new epoch, we add 100 rewards.
- For the rest of the epoch, we accrue 99.99 rewards.
- The check is always false, the 99 rewards will not be added to this epoch, despite having accrued them during this epoch.

Recommendation: Document the change or consider always calling `notifyRewardAmount()` when `fees / DURATION > 0`

Velodrome: I think this issue is a bit challenging to address as fee generation is unpredictable as could be influenced by exogenous factors. In general, bribes and emissions will be fairly consistent on a week-to-week basis, but fees could spike one week (due to a material market event resulting in increased interest) and then decline in other weeks.

I think the code as it is helps smooth fees across epochs (as it prevents too many fees from accumulating over a single epoch), so the best we can do is likely document the change.

Spearbit: Acknowledged.

5.4.4 Not synced with `Epochs`

Severity: Low Risk

Context: Gauge.sol#L209

Description: If there's enough delays in calling the `notifyRewardAmount()` function, a full desync can happen.

Recommendation: It may be ideal to queue rewards to the next epoch to avoid any form of grief/race condition. That said, this may create negative externalities.

So perhaps the check could be

- If in the first half of the epoch, queue rewards to the current epoch.
- If in the second half of the epoch, queue rewards to the next epoch.

Velodrome: I am worried about edge cases where a pool may fall into disuse and then be used again. It looks like we can mitigate this by setting `periodFinish` to the end of the epoch / start of the next epoch. I am aware that this will encourage users calls to `Voter.distribute()` late but the plan for this function is to automate it so that it is called at the beginning of every epoch.

Spearbit: Agree that it makes sense to align `periodFinish` with the start of the following epoch to avoid any time drift. But I think this change would actually make things more unfair as like what was stated above, users are incentivized to call `Voter.distribute()` late into the epoch to maximize `rewardRate`.

Velodrome: Voter.distribute() is a target for keeper automation, which will make it less likely that it will be called late. It appears that `periodFinish` could slightly inflate gauge rewards if the second `notifyRewardAmount` call is closer to the start of its' relative epoch than the prior `notifyRewardAmount`. How?

- Assume `notifyRewardAmount` (epoch 1) is called 10 seconds after the start of epoch 1.
  - `periodFinish` is now set to 10 seconds after the start of epoch 2
- `notifyRewardAmount` (epoch 2) is called 1 second after the start of epoch 2
  - `timestamp < periodFinish` by 9 seconds, therefore `rewardRate` adds the leftover and the amount deposited.

I believe what would be best is to always set `periodFinish` to the epoch start of the next epoch so that this inflation does not happen.

Velodrome: Fixed in commit a336f7.

Spearbit: Verified.

5.4.5 Dust losses in `notifyRewardAmount`

Severity: Low Risk

Context: Gauge.sol#L196

Description: This should cause dust losses which are marginal but are never queued back. See private link to code-423n4/2022-10-3xcalibur-findings#410.

Vs SNX implementation which does queue the dust back.

Users may be diluted by distributing the `_leftover` amount of another epoch period of length DURATION if the total supply deposited in the gauge continues to increase over this same period. On the flip side, they may also benefit if users withdraw funds from the gauge during the same epoch.

Recommendation: Re-queuing the `totalBalance` on each notifyRewardAmount should allow re-using those dust amounts.

Additionally, these dust amounts are going to be hard to deal with because you may have rewards from older epochs, meaning that re-queuing the balance on the contract would cause a double-spend Napkin math suggests the loss be very marginal assuming: WEEK = 60*60*24*7 GAUGES = 50 GAUGES = 52 WEEK * GAUGES * GAUGES = 1572480000

1572480000 / 1e18 = 1.57248e-7

Velodrome: I was under the impression that re-queueing `balanceOf` would cause losses as not all emissions will necessarily be claimed.

Spearbit:

> I was under the impression that re-queueing `balanceOf` would cause losses as not all emissions will necessarily be

Yes, that's probably a bigger issue than the marginal rounding

Recommend you check dust on a few of the most used gauges, if it's in marginal amounts, it's prob not worth fixing

Velodrome: It is difficult to check with the prior gauges as there are always unclaimed rewards, but following the math and given that the token will always be 18 decimals, and the divisor always <= 604800 (seconds in a week), the losses will be minimal.

Spearbit: Acknowledged.

5.4.6 All rewards are lost until `Gauge` or `Bribe` deposits are non-zero

Severity: Low Risk

Context: Gauge.sol#L123-L128

Description: Flagging this old finding which is still valid for all SNX like gauges. See private link to code-423n4/2022-10-3xcalibur-findings#429.

Because the rewards are emitted over `DURATION`, if no deposit has happened and `notifyRewardAmount()` is called with a non-zero value, all rewards will be forfeited until `totalSupply` is non-zero as nobody will be able to claim them.

Recommendation: Document this risk to end users and tell them to deposit before voting on a gauge.

Velodrome: To confirm the issue as I cannot see the findings If a gauge has no LP tokens deposited, and `notifyRewardAmount` is called with a non-zero value, some amount of tokens will be trapped as no one can claim them.

Will add documentation. I think in general users will deposit if there is potential emissions to collect (as the voting occurs in the week prior, and the emissions only get notified the following week).

Spearbit: Acknowledged.

5.4.7 Difference in `getPastTotalSupply` and `propose`

Severity: Low Risk

Context: VeloGovernor.sol#L45-L48

Description: The `getPastTotalSupply()` function currently uses `block.number`, but OpenZeppelin's `propose()` function will use votes from `block.timestamp - 1` as seen here.

This could enable

- Front-run and increase total supply to cause proposer to be unable to `propose()`.
- Require higher tokens than expected if total supply can grow within one block.

Proposals could be denied as long as a whale is willing to lock more tokens to increase the total supply and thereby increase the proposal threshold.

Recommendation: Consider computing total supply and votes values in the same block.

Velodrome: Fixed in commit 08c2bcc.

Spearbit: Verified.

5.4.8 Delaying `update_period` may award more emissions

Severity: Low Risk

Context: Minter.sol#L87

Description: First nudge can be performed on the first tail period, delaying `update_period()` may award more emissions, because of the possible delay with the first proposal, waiting to call `update_period()` will allow the use of the updated nudged value.

This is marginal (1BPS in difference)

Recommendation: Consider documenting this behavior or enforcing a separation between `nudge` and `updated_period`

Velodrome: To clarify, the issue is that `update_period` can be delayed until after nudge, thus allowing for potentially more (or less) emissions?

Documentation is fine, but I will also note that `update_period` is a target for keeper automation which should reduce the likelihood of this.

Spearbit: Acknowledged. Additionally, you could end up `nudge`ing before or after calling `update_period` which would impact 1 BPS in emissions.

5.4.9 Incorrect math for future factories and pools

Severity: Low Risk

Context: Router.sol#L673-L700

Description: Because `quoteLiquidity()` assumes an `x * y = k` formula, its quote value will be incorrect when using a custom factory that uses a different curve.

```
//Router.sol#L673-L700
function _quoteZapLiquidity(
    address tokenA,
    address tokenB,
    bool stable,
    address _factory,
    uint256 amountADesired,
    uint256 amountBDesired,
    uint256 amountAMin,
    uint256 amountBMin
) internal view returns (uint256 amountA, uint256 amountB) {
    require(amountADesired >= amountAMin);
    require(amountBDesired >= amountBMin);
    (uint256 reserveA, uint256 reserveB) = getReserves(tokenA, tokenB, stable, _factory);
    if (reserveA == 0 && reserveB == 0) {
        (amountA, amountB) = (amountADesired, amountBDesired);
    } else {
        uint256 amountBOptimal = quoteLiquidity(amountADesired, reserveA, reserveB);
        if (amountBOptimal <= amountBDesired) {
            require(amountBOptimal >= amountBMin, "Router: insufficient B amount");
            (amountA, amountB) = (amountADesired, amountBOptimal);
        } else {
            uint256 amountAOptimal = quoteLiquidity(amountBDesired, reserveB, reserveA);
            assert(amountAOptimal <= amountADesired);
            require(amountAOptimal >= amountAMin, "Router: insufficient A amount");
            (amountA, amountB) = (amountAOptimal, amountBDesired);
        }
    }
}
```

The math may be incorrect for future factories and pools, while the current math is valid for `x * y = k`, any new AMM math (e.g Bounded / V3 math, Curve V2, Oracle driven AMMs) may turn out to be incorrect.

This may cause issues when performing zaps with custom factories.

Recommendation: Consider whether custom factories should use zaps and consider extending the code to allow custom factories to specify their own `quoteLiquidity()` implementation.

Velodrome: We are aware of this issue. Future factories will have different parameters for swaps / adding and removing liquidity and will likely require a different router, so a separate router will be created for those.

I guess we can make it clearer in the documentation that this will be used exclusively for the current stable/volatile pair implementation.

Indeed, the router implementation will evolve based on the pool implementation. I agree that docs should communicate this clearly.

Spearbit: Acknowledged.

5.4.10 Add function to remove whitelisted token and NFT

Severity: Low Risk

Context: Voter.sol

Description: In the `Voter` contract, the governor can only add tokens and NFTs to the whitelist array. However, it is missing the functionality to remove whitelisted tokens and NFTs. If any whitelisted token or NFT has an issue, it cannot be removed from the list.

Recommendation: It is recommended to add the remove functionality, which could be done by the following change:

```
- function whitelistToken(address _token) external {
+ function whitelistToken(address _token, bool _enable) external {
  require(_msgSender() == governor, "Voter: not governor");
-  _whitelistToken(_token);
+  _whitelistToken(_token, _enable);
 }

- function _whitelistToken(address _token) internal {
+ function _whitelistToken(address _token, bool _enable) internal {
-  require(!isWhitelistedToken[_token], "Voter: token already whitelisted");
-  isWhitelistedToken[_token] = true;
-  emit WhitelistToken(_msgSender(), _token);

+  isWhitelistedToken[_token] = _enable;
+  emit WhitelistToken(_msgSender(), _token, _enable);
 }

- function whitelistNFT(uint256 _tokenId) external {
+ function whitelistNFT(uint256 _tokenId, bool _enable) external {
    address _sender = _msgSender();
    require(_sender == governor, "Voter: not governor");
-   require(!isWhitelistedNFT[_tokenId], "Voter: nft already whitelisted");
-   isWhitelistedNFT[_tokenId] = true;
+   isWhitelistedNFT[_tokenId] = _enable;
-   emit WhitelistNFT(_sender, _tokenId);
+   emit WhitelistNFT(_sender, _tokenId, _enable);
 }
```

Velodrome: This issue has been fixed in commit 2ace8b.

Spearbit: Verified.

5.4.11 Unnecessary approve in `Router`

Severity: Low Risk

Context: Router.sol#L656-L657, Router.sol#L712

Description: The newly added Zap feature uses max approvals, which are granted to pairs.

However, the `Pair` contract does not pull tokens from the router, and therefore unnecessarily calls `approve()` in the router.

Because of the possibility of specifying a custom factory, attackers will be able to set up approvals from any token to their contracts.

This may be used to scam end-users, for example by performing a swap on these malicious factories.

Recommendation: While no attack was identified, the usage of max approvals may be too liberal. A more cautious approach would be to set the allowance to the exact value necessary and then reset it back to 0.

Alternatively, do not give any allowance to the pair.

Velodrome: Fixed in commit c799c6.

Spearbit: Verified.

5.4.12 The `current` value of a `Pair` is not always returning a 30-minute TWAP and can be manipulated.

Severity: Low Risk

Context: Pair.sol#L276-L288

Description: The `current` function returns a current TWAP. It fetches the last `observation` and calculates the TWAP between the last observation. The `observation` is pushed every thirty minutes. However, the interval between current timestamp and the last observation varies a lot. In most cases, the TWAP interval is shorter than 30 minutes.

```
//Pair.sol#L284-L288
uint256 timeElapsed = block.timestamp - _observation.timestamp;
@audit: timeElapsed can be much smaller than 30 minutes.
uint256 _reserve0 = (reserve0Cumulative - _observation.reserve0Cumulative) / timeElapsed;
uint256 _reserve1 = (reserve1Cumulative - _observation.reserve1Cumulative) / timeElapsed;
amountOut = _getAmountOut(amountIn, tokenIn, _reserve0, _reserve1);
```

If the last observation is newly updated, the `timeElapsed` will be much shorter than 30 minutes. The cost of price manipulation is cheaper in this case.

Assume the last observation is updated at `T`. The exploiter can launch an attack at `T + 30_MINUTES - 1`

1. At `T + 30_MINUTES - 1`, the exploiter tries to manipulate the price of the `pair`. Assume the price is moved to 100x.
2. At `T + 30_MINUTES`, the exploiter pokes the pair. The pair push an `observation` with the price = 100x.
3. At `T + 30_MINUTES + 1`, the exploiter tries to exploit external protocols. The `current` function fetches the last observation and calculates the TWAP between the last observation and the current price. It ends up calculating the two-block-interval TWAP.

Recommendation: There are two possible mitigations

1. Check whether external protocols are using the `current` function. We shall document and inform external protocols of this potential risk.
2. If the `current` function is not popular, consider removing this function.

An alternative solution is a `lastTWAP` where always calculate the TWAP based on the last two observations. As the time elapsed between two observations is always larger than 30 minutes, the manipulation cost is guaranteed to be high.

It's worth mentioning that the cost of multiple blocks MEV and manipulation varies among different chains. The trustworthiness of TWAP relies on the block producer/sequencer, and the mechanism is not transparent on many Layer 2 and alternative Layer 1. Also, two blocks manipulation is believed to be practical on Ethereum. It's likely that all L2 chains will move toward this model. chainsecurity.com/oracle-manipulation-after-merge As a result, it's recommended to be conservative when using a TWAP.

Velodrome: I think I am okay with removing `current` to prevent other protocols from using it inappropriately. Will check with the team.

Fixed in commit fc936a.

Spearbit: Verified.

5.4.13 Calculation error of `getAmountOut` leads to revert of Router

Severity: Low Risk

Context: Pair.sol#L450-L476

Description: The function `getAmountOut` in `Pair` calculates the correct swap amount and token price.

```
//Pair.sol#L442-L444
function _f(uint256 x0, uint256 y) internal pure returns (uint256) {
    return (x0 * ((((y * y) / 1e18) * y) / 1e18)) / 1e18 + (((((x0 * x0) / 1e18) * x0) / 1e18) * y) /
    ↪  1e18;
}
```

```
//Pair.sol#L450-L476
function _get_y(
    uint256 x0,
    uint256 xy,
    uint256 y
) internal pure returns (uint256) {
    for (uint256 i = 0; i < 255; i++) {
        uint256 y_prev = y;
        uint256 k = _f(x0, y);
        if (k < xy) {
            uint256 dy = ((xy - k) * 1e18) / _d(x0, y);
            y = y + dy;
        } else {
            uint256 dy = ((k - xy) * 1e18) / _d(x0, y);
            y = y - dy;
        }
        if (y > y_prev) {
            if (y - y_prev <= 1) {
                return y;
            }
        } else {
            if (y_prev - y <= 1) {
                return y;
            }
        }
    }
    return y;
}
```

The `getAmountOut` is not always correct. This results in the router unexpectedly revert a regular and correct transaction.

We can find one parameter that the router will fail to swap within 5s fuzzing.

```
function testAmountOut(uint swapAmount) public {
    vm.assume(swapAmount < 1_000_000_000 ether);
    vm.assume(swapAmount > 1_000_000);
    uint256 reserve0 = 100 ether;
    uint256 reserve1 = 100 ether;
    uint amountIn = swapAmount - swapAmount * 2 / 10000;
    uint256 amountOut = _getAmountOut(amountIn, token0, reserve0, reserve1);
    uint initialK = _k(reserve0, reserve1);
    reserve0 += amountIn;
    reserve1 -= amountOut;
    console.log("initial k:", initialK);
    console.log("curent k:", _k(reserve0, reserve1));
    console.log("curent smaller k:", _k(reserve0, reserve1 - 1));
    require(initialK < _k(reserve0, reserve1), "K");
    require(initialK > _k(reserve0, reserve1-1), "K");
}
```

After the fuzzer have a counter example of `swapAmount = 1413611527073436` We can test that the `Router` will revert if given the fuzzed params.

```
contract PairTest is BaseTest {
  function testRouterSwapFail() public {
        Pair pair = Pair(factory.createPair(address(DAI), address(FRAX), true));
        DAI.approve(address(router), 100 ether);
        FRAX.approve(address(router), 100 ether);
        _addLiquidityToPool(
            address(this),
            address(router),
            address(DAI),
            address(FRAX),
            true,
            100 ether,
            100 ether
        );
        uint swapAmount = 1413611527073436;
        DAI.approve(address(router), swapAmount);
        // vm.expectRevert();
        console.log("fee:", factory.getFee(address(pair), true));
        IRouter.Route[] memory routes = new IRouter.Route[](1);
        routes[0] = IRouter.Route(address(DAI), address(FRAX), true, address(0));
        uint daiAmount = DAI.balanceOf(address(pair));
        uint FRAXAmount = FRAX.balanceOf(address(pair));
        console.log("daiAmount: ", daiAmount, "FRAXAmount: ", FRAXAmount);
        vm.expectRevert("Pair: K");
        router.swapExactTokensForTokens(swapAmount, 0, routes, address(owner), block.timestamp);

    }
}
```

Recommendation: There are two causes of the miscalculation

1. The function `_f` gets a different value from `_k` because of the rounding error.

```
+        uint256 _a = (x0 * y) / 1e18;
+        uint256 _b = ((x0 * x0) / 1e18 + (y * y) / 1e18);
+        return (_a * _b) / 1e18;
-         return (x0 * ((((y * y) / 1e18) * y) / 1e18)) / 1e18 + (((((x0 * x0) / 1e18) * x0) / 1e18) *
↪   y) / 1e18;
```

2. `dy` at Pair.sol#L459 will get screwed by the rounding error.

```
function _get_y(
       uint256 x0, // @audit (amountIn + reserveA) post reserveA
       uint256 xy, // k
       uint256 y // reserveB
   ) internal view returns (uint256) {
       for (uint256 i = 0; i < 255; i++) {
           uint256 y_prev = y;
           //@audit _f have a different rounding to _k
           uint256 k = _f(x0, y);
           if (k < xy) {
               //@audit: there are two cases where dy == 0
               // case 1: The y is converged and we find the correct answer
               // case 2: _d(x0, y) is too large compare to (xy - k) and the rounding error
               //         screwed us.
               //         In this case, we need to increase y by 1
               uint256 dy = ((xy - k) * 1e18) / _d(x0, y);
               if (dy == 0) {
                   if (k == xy ) {
                       // We found the correct answer. Return y
                       return y;
                   }
                   if (_k(x0, y + 1) > xy) {
                       // If _k(x0, y + 1) > xy, then we are close to the correct answer.
                       // There's no closer answer than y + 1
                       return y + 1;
                   }
                   dy = 1;
               }
               y = y + dy;
           } else {
               uint256 dy = ((k - xy) * 1e18) / _d(x0, y);
               if (dy == 0) {
                   if (k == xy || _f(x0, y - 1) < xy) {
                       // Likewise, if k == xy, we found the correct answer.
                       // If _f(x0, y - 1) < xy, then we are close to the correct answer.
                       // There's no closer answer than "y"
                       // It's worth mentioning that we need to find y where f(x0, y) >= xy
                       // As a result, we can't return y - 1 even it's closer to the correct answer
                       return y;
                   }
                   dy = 1;
               }
               y = y - dy;
           }
       }
       // @audit - should never happen. If it does, it means it doesn't converge for 255 iterations
       // @audit - should assign a custom error to save gas.
       revert("y not found");
   }
```

Velodrome: I have reviewed this and I agree with the finding. We can fix it for the new `Pair` contracts but note that like the *"`current` value of a Pair is not always returning a 30-minute TWAP and can be manipulated"* issue, it will only be fixed for new Pair contracts, while old Pair contracts will continue to have the faulty code in it.

Fixed in commit 9ca981.

Spearbit: Verified.

5.4.14 `VotingEscrow` checkpoints is not synchronized

Severity: Low Risk

Context: VotingEscrow.sol#L1309-L1351, VotingEscrow.sol#L1364-L1415

Description: Delegating token ids is not synchronizing correctly `fromBlock` variable in the checkpoint, by leaving it not updated the functions `getPastVotesIndex` and `_findWhatCheckpointToWrite` could return the incorrect index.

Recommendation: It is recommended to update the `fromBlock` variable of checkpoint in functions `_moveTokenDelegates` and `_moveAllDelegates`

Velodrome: Fixed in commit a670bf.

Spearbit: Verified.

5.4.15 Wrong proposal expected value in `VeloGovernor`

Severity: Low Risk

Context: VeloGovernor.sol#L14-L16

Description: The expected values of `MAX_PROPOSAL_NUMERATOR` and `proposalNumerator` are incorrect, in the current implementation max proposal is set to 0.5%, the expected value is 5%, and the proposal numerator starts at 0.02%, and not at 0.2% as expected.

Recommendation: It is recommended to fix the value to

```
- uint256 public constant MAX_PROPOSAL_NUMERATOR = 50; // max 5%
+ uint256 public constant MAX_PROPOSAL_NUMERATOR = 500; // max 5%
  uint256 public constant PROPOSAL_DENOMINATOR = 10_000;
- uint256 public proposalNumerator = 20; // start at 0.02%
+ uint256 public proposalNumerator = 2; // start at 0.02%
```

Velodrome: Fixed as part of this commit 64fe60.

Spearbit: Verified.

5.4.16 `_burn` function will always revert if the caller is the approved spender

Severity: Low Risk

Context: VotingEscrow.sol#L556

Description: The owner or the approved spender is allowed to trigger the `_burn` function. However, whenever an approved spender triggers this function, it will always revert. This is because the `_removeTokenFrom` function will revert internally if the caller is not the owner of the NFT as shown below.

```
function _removeTokenFrom(address _from, uint256 _tokenId) internal {
    // Throws if `_from` is not the current owner
    assert(idToOwner[_tokenId] == _from);
```

As a result, an approved spender will not be able to withdraw or merge a veNFT on behalf of the owner because the internal `_burn` function will always revert.

Recommendation: Update the `_burn` function to pass the owner's address instead of the caller's address to the `_removeTokenFrom` function

```
 function _burn(uint256 _tokenId) internal {
    require(_isApprovedOrOwner(msg.sender, _tokenId), "VotingEscrow: caller is not owner nor approved");
    address owner = ownerOf(_tokenId);
    ..SNIP..
    // Remove token
-   _removeTokenFrom(msg.sender, _tokenId);
+   _removeTokenFrom(owner, _tokenId);
    emit Transfer(owner, address(0), _tokenId);
 }
```

Velodrome: Fixed in commit 7d5a78.

Spearbit: Verified.

5.5 Gas Optimization

5.5.1 OpenZeppelin's `Clones` library can be used to cheaply deploy rewards contracts

Severity: Gas Optimization

Context: VotingRewardsFactory.sol, ManagedRewardsFactory.sol, PairFactory.sol#L153, GaugeFactory.sol

Description: OpenZeppelin's `Clones` library allows for significant gas savings when there are multiple deployments of the same family of contracts. This would prove useful in several factory contracts which commonly deploy the same type of contract.

Minimal proxies make use of the same code even when initialization data may be different for each instance. By pointing to an implementation contract, we can delegate all calls to a fixed address and minimise deployment costs.

Recommendation: Consider making use of this library in any of the factory contracts.

Velodrome: At this stage, we will make this change for the Pair contract as it appears to get the most benefit. For other contracts, we will consider it but it is a low priority.

Fix for pairs in commit cd4698.

Spearbit: Verified.

5.5.2 `VelodromeTimeLibrary` functions can be made `unchecked`

Severity: Gas Optimization

Context: VelodrometimeLibrary.sol#L7-L14

Description: Running the following fuzz test

```
pragma solidity 0.8.13;

import "forge-std/Test.sol";

contract VelodromeTimeLibrary {
    uint256 public constant WEEK = 7 days;

    /// @dev Returns start of epoch based on current timestamp
    function epochStart(uint256 timestamp) public pure returns (uint256) {
        unchecked {
            return timestamp - (timestamp % WEEK);
        }
    }

    /// @dev Returns unrestricted voting window
    function epochEnd(uint256 timestamp) public pure returns (uint256) {
        unchecked {
```

```
            return timestamp - (timestamp % WEEK) + WEEK - 1 hours;
        }
    }
}

contract VelodromeTimeLibraryTest is Test {
    VelodromeTimeLibrary vtl;
    uint256 public constant WEEK = 7 days;

    function setUp() public {
        vtl = new VelodromeTimeLibrary();
    }

    function testEpochStart(uint256 timestamp) public {
        uint256 uncheckedVal =  vtl.epochStart(timestamp);
        uint256 normalVal = timestamp - (timestamp % WEEK);
        assertEq(uncheckedVal, normalVal);
    }

    function testEpochEnd(uint256 timestamp) public {
        uint256 uncheckedVal = vtl.epochEnd(timestamp);
        uint256 normalVal = timestamp - (timestamp % WEEK) + WEEK - 1 hours;
        assertEq(uncheckedVal, normalVal);
    }
}
```

One can see that both `VelodromeTimeLibrary` functions will only start to overflow at a ridiculously high `timestamp` input.

Recommendation: With that in mind it would be safe to consider making both `VelodromeTimeLibrary` functions `unchecked`.

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.5.3 Skip call can save gas

Severity: Gas Optimization

Context: Voter.sol#L430-L434

Description: `distribute(address[] memory _gauges)` is meant to be used for multiple gauges but it calls `minter.update_period` before each call to `notifyRewardAmount`

Recommendation: By refactoring to call `update_period` only once, you can save at least 200 gas per additional gauge in the list.

Velodrome: `distribute(address _gauge)` exists in the case there are too many gauges where `distribute(address[] memory _gauges)` would run out of gas when distributing to all gauges at once. However, I do believe `distribute(address _gauge)` can be turned internal where `distribute(address[] memory _gauges)` only calls `update_period()` once before the looped call of `distribute()`.

Spearbit: Acknowledged.

5.5.4 Change to zero assignment to save gas

Severity: Gas Optimization

Context: Voter.sol#L141

Description: It is not necessary to subtract the total value from the votes instead you should set it directly to zero.

Recommendation: It is recommended to set votes to zero

```
- votes[_tokenId][_pool] -= _votes;
+ delete votes[_tokenId][_pool];
```

Velodrome: Fixed in commit 111d83.

Spearbit: Verified.

5.5.5 Refactor to skip an `SLOAD`

Severity: Gas Optimization

Context: VotingEscrow.sol#L831-L832

Description: It is possible to skip an SLOAD by refactoring the code as it is in recommendation.

Recommendation: It is recommended to change the addition of tokenId to

```
- ++tokenId;
- uint256 _tokenId = tokenId;
+ uint256 _tokenId = ++tokenId;
```

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.5.6 Tail variable can be removed to save gas

Severity: Gas Optimization

Context: Minter.sol#L43

Description: It is possible to save gas by freeing the tail slot, which can be replaced by check `weekly < TAIL_START`

Recommendation: You can replace the tail variable by using the check

```
- bool public tail;

 function nudge() external {
  ...
-  require(tail, "Minter: not in tail emissions yet");
+  require(weekly < TAIL_START, "Minter: not in tail emissions yet");
  ...
 }

 function update_period() external returns (uint256 _period) {
  ...
-  bool _tail = tail;
+  bool _tail = weekly < TAIL_START;
  ...
 }
```

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.5.7 Use a bitmap to store nudge `proposals` for each epoch

Severity: Gas Optimization

Context: Minter.sol#L41

Description: The usage of a bitmap implementation for boolean values can save a significant amount of gas. The `proposals` variable can be indexed by each epoch which should only increment once per week.

Recommendation: Consider making use of a bitmap implementation to efficiently pack boolean values.

Velodrome: Addressed.

Spearbit: The team created a PR to implement the suggestion.

5.5.8 `isApproved` function optimization

Severity: Gas Optimization

Context: FactoryRegistry.sol#L63-L69

Description: Because settings are all known, you could do an if-check in memory rather than in storage, by validating first the fallback settings. The recommended implementation will become cheaper for the base case, negligibly more expensive in other cases ~10s of gas

Recommendation: Change `isApproved` function to follow

```
 function isApproved(
 address pairFactory,
 address votingRewardsFactory,
 address gaugeFactory
    ) external view returns (bool) {
+   if ((pairFactory == fallbackPairFactory) &&
+         (votingRewardsFactory == fallbackVotingRewardsFactory) &&
+         (gaugeFactory == fallbackGaugeFactory) {
+               return true;
+   }
    return _approved[pairFactory][votingRewardsFactory][gaugeFactory];
 }
```

By doing this change this check would also be redundant

```
 function unapprove(
    address pairFactory,
    address votingRewardsFactory,
    address gaugeFactory
 ) external onlyOwner {
  require(_approved[pairFactory][votingRewardsFactory][gaugeFactory], "FactoryRegistry: not approved");
- require(
-     !((pairFactory == fallbackPairFactory) &&
-         (votingRewardsFactory == fallbackVotingRewardsFactory) &&
-         (gaugeFactory == fallbackGaugeFactory)),
-     "FactoryRegistry: Cannot delete the fallback route"
- );
  delete _approved[pairFactory][votingRewardsFactory][gaugeFactory];
  emit Unapprove(pairFactory, votingRewardsFactory, gaugeFactory);
 }
```

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.5.9 Use calldata instead of memory to save gas

Severity: Gas Optimization

Context: VotingRewardsFactory.sol#L9, Voter.sol#L96

Description: Using `calldata` avoids copying the value into memory, reducing gas cost

Recommendation: Change function variables array from `memory` to `calldata`

```
//VotingRewardsFactory
- function createRewards(address[] memory rewards)
+ function createRewards(address[] calldata rewards)

//Voter
- function initialize(address[] memory _tokens, address _minter) external {
+ function initialize(address[] calldata _tokens, address _minter) external {
```

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.5.10 Cache store variables when used multiple times

Severity: Gas Optimization

Context: SinkManager.sol

Description: Storage loads are very expensive compared to memory loads, storage values that are read multiple times should be cached avoiding multiple storage loads.

In `SinkManager` contract use multiple times the storage variable `ownedTokenId`

Recommendation: Cache storage variables that are used multiple times:

```
 function convertVELO(uint256 amount) external {
+ uint256 _ownedTokenId = ownedTokenId;
-  require(ownedTokenId != 0, "SinkManager: tokenId not set");
+  require(_ownedTokenId != 0, "SinkManager: tokenId not set");
   ...
-  ve.increase_amount(ownedTokenId, amount);
+  ve.increase_amount(_ownedTokenId, amount);
     ...
 }
```

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.5.11 Add immutable to variable that don't change

Severity: Gas Optimization

Context: SinkConverter.sol#L13-L15, SinkManager.sol#L31-L40, FactoryRegistry.sol#L16-L18, Gauge.sol#L24

Description: Using `immutable` for variables that do not changes helps to save on gas used. The reason has been that `immutable` variables do not occupy a storage slot when compiled, they are saved inside the contract byte code.

Recommendation: Add `immutable` keyword for each one of the variables in context

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.5.12 Use Custom Errors

Severity: Gas Optimization

Context: Across All Contracts

Description: As one can see here:

> "there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., revert("Insufficient funds.");), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them."

Recommendation: Consider using Custom Errors instead of using error strings, to reduce deployment and runtime cost. This would save both deployment and runtime cost.

Velodrome: Partially addressed in commit 9b1f6e for the main contracts.

Spearbit: Verified.

5.5.13 Cache array length outside of loop

Severity: Gas Optimization

Context: Pair.sol#L298, RewardsDistributor.sol#L289, Router.sol#L122, Router.sol#L401, Router.sol#L491, Router.sol#L531, Router.sol#L549, Voter.sol#L98, Voter.sol#L335, Voter.sol#L373, Voter.sol#L385, Voter.sol#L397, Voter.sol#L403, Voter.sol#L431, VotingEscrow.sol#L1242, VotingEscrow.sol#L1293, VotingEscrow.sol#L1323, VotingEscrow.sol#L1342, VotingEscrow.sol#L1379, VotingEscrow.sol#L1402, GovernorSimple.sol#L336, GovernorSimple.sol#L353, Reward.sol#L237, VotingReward.sol#L11

Description: If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Recommendation: Consider simply doing something like so before the for loop: `uint length = variable.length`. Then add `length` in place of `variable.length` in the for loop.

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6 Informational

5.6.1 Withdrawing from a managed `veNFT` locks the user's `veNFT` for the maximum amount of time

Severity: Informational

Context: VotingEscrow.sol#L177-L186

Description: A user may deposit their `veNFT` through the `depositManaged()` function with any unlock time value. However, upon withdrawing, the unlock time is automatically configured to `(block.timestamp + MAXTIME / WEEK) * WEEK`. This is poor UX and it does not give users much control over the expiry time of their `veNFT`.

Recommendation: While this may be an intentional design decision, it does not give users much autonomy over their own `veNFT` after depositing into a managed `veNFT`. Ensure this is documented and well-understood by all users interacting with the managed `veNFT` mechanism.

Velodrome: This is an intentional design decision and will be made explicit to users in the interface prior to interacting with managed NFTs. Added comments in commit eb7f0e.

Spearbit: Acknowledged.

5.6.2 `veNFT` split functionality can not be disabled

Severity: Informational

Context: VotingEscrow.sol#L1186-L1189

Description: Once split functionality has been enabled via the `enableSplitForAll()`, it is not possible to disable this feature in the future. It does not pose any additional risk to have it disabled once users have already split their `veNFTs` because the protocol allows for these locked amounts to be readily withdrawn upon expiry.

Recommendation: Ensure this is documented or consider adding a `disableSplitForAll()` function, callable only by the Velodrome team.

Velodrome: Fixed in commit 927cdb.

Spearbit: Verified.

5.6.3 Anyone can notify the `FeesVotingReward` contract of new rewards

Severity: Informational

Context: Gauge.sol#L62-L88, BribeVotingReward.sol, Voter.sol#L274-L277

Description: While the `BribeVotingReward` contract intends to allow bribes from anyone, the `FeesVotingReward` contract is designed to receive fees from just the `Gauge` contract. This is inconsistent with other reward contracts like `LockedManagedReward`.

Recommendation: Ensure this is acceptable behaviour and consider documenting this within `FeesVotingReward`. Alternatively, it may be worthwhile restricting the `notifyRewardAmount()` function to be callable only from the `Gauge` contract.

Velodrome: I think in line with making permissions as restrictive as possible, we can go with restricting `notifyRewardAmount` for `FeesVotingReward` to the gauge only. Something like

```
require(IVoter(voter).gaugeToFees(sender) == address(this));
```

Velodrome: Fixed in commit 8de67e.

Spearbit: Verified.

5.6.4 Missing check in `merge` if the `_to` NFT has voted

Severity: Informational

Context: VotingEscrow.sol#L1130

Description: The `merge()` function is used to combine a `_from` VeNFT into a `_to` veNFT. It starts with a check on if the `_from` VeNFT has voted or not. However, it doesn't check if the `_to` VeNFT has voted or not. This will cause the user to have less voting power, leaving rewards and/or emissions on the table, if they don't call `poke() || reset()`.

Although this would only be an issue for an unaware user. An aware user would still have to waste gas on either of the following:

1. An extra call to `poke() || reset()`.
2. Vote with the `_to` veNFT and then call `merge()`.

Recommendation: Consider adding the following check:

```
require(!voted[_to], "VotingEscrow: voted");
```

Velodrome: This is a known issue. We removed the requirement on _to as it creates a worse user experience (e.g. you may vote and then realize you want to merge but can't and have to wait until the next epoch). By not constraining _to, the user has a few more options, with the trade-off being they must call poke() if they merge

or risk losing some rewards that epoch, we also made the voting (Voter::vote()) to be much more flexible in that regard. Anybody increasing their veNFT lock amount or merging into their veNFT, or generally those who want to change their vote last minute, are free to do it. V2 front-end will communicate the need to re-cast the votes after:

- `merge()`
- `increaseAmount()`
- `increaseUnlockTime()`

Regarding poke, we will not be adding it to VotingEscrow due to gas concerns. I think regarding this issue, we will leave it as is, as it provides greater flexibility on how users can use their veNFTs. Confirmed with the team.

Spearbit: Acknowledged.

5.6.5 Ratio of invariant `_k` to `totalSupply` of the AMM pool may temporarily decrease

Severity: Informational

Context: Pair.sol#L365-L386

Description: The `burn` function directly sends the reserve pro-rated to the liquidity token. This is a simple and elegant way. Nevertheless, two special features of the current AMM would lead to a weird situation.

1. The `fee` of the AMM pool is sent to the fee contract instead of being absorbed into the pool;
2. The stable pool's curve x3y + y3x have a larger rounding error compare to uni-v2's constant product formula.

The invariant K in a stable pool can decrease temporarily when a user performs certain actions like minting a token, doing a swap, and withdrawing liquidity. This means that the ratio of K to the total supply of the pool is not monotonously increasing. In most cases, this temporary decrease is negligible and the ratio of K to the total supply of the pool will eventually increase again. However, the ratio of K to the total supply is an important metric for calculating the value of LP tokens, which are used in many protocols. If these protocols are not aware of the temporary decrease in the K value, they may suffer from serious issues (e.g. overflow).

The root cause of this issue is: there are always rounding errors when using "balance" to calculate invariant k. Sometimes, the rounding error is larger. if an lp is minted when the rounding error is small (ratio of amount: k is small) and withdrawn when the rounding error is large (ratio of amount: k is large). The total invariant decreased.

We can find a counter-example where the invariant decrease.

```
function testRoundingErrorAttack(uint swapAmount) public {
// The counter-example: swapAmount = 52800410888861351333
vm.assume(swapAmount < 100_000_000 ether);
vm.assume(swapAmount > 10 ether);
uint reserveA = 10 ether;
uint reserveB = 10 ether;

uint initialK = _k(reserveA, reserveB);
reserveA *= 2;
reserveB *= 2;
uint tempK = _k(reserveA, reserveB);
reserveB -= _getAmountOut(swapAmount, token0, reserveA, reserveB);
reserveA += swapAmount;
vm.assume(tempK <= _k(reserveA, reserveB));

reserveA -= reserveA / 2;
reserveB -= reserveB / 2;

require(_k(reserveA, reserveB) >  initialK, "Rounding error attack!");
}
```

Recommendation: We recommend documenting this issue. The ratio of invariant `k` to `totalSupply` is a common way to securely value lp tokens. This may potentially break external protocols (e.g. overflow when evaluating rewards distribution in a vault)

There are rounding errors everywhere; this in most cases, would not be critical. The `burn` function of the AMM pool, while bearing some rounding error, is an efficient way to return liquidity. Also, modifying the mechanism of burning liquidity in an AMM pool would possibly lead to serious issues.

Velodrome: Will add documentation around this issue.

Spearbit: Acknowledged.

5.6.6 Inconsistent check for adding value to a lock

Severity: Informational

Context: VotingEscrow.sol#L807

Description: `depositFor` allows anyone to add value to an existing lock

However `increaseAmount`, which for `NORMAL` locks is idempotent, has a check to only allow an approved or Owner to increase the amount.

Recommendation: Document the inconsistency, or decide if the check should be performed or removed from both functions

Velodrome: We're considering keeping the `depositFor` open and just documenting it's purpose.

Spearbit: Acknowledged.

5.6.7 Privileged actors are incentivized to front-run each other

Severity: Informational

Context: Voter.sol#L209-L224

Description: Privileged actors are incentivized to front-run each other and vote at the last second, because of the FIFS OP sequencer, managers will try to vote exactly at the penultimate block in order to maximize their options (voting can only be done once)

Recommendation: Velodrome: These actions already exist in the current protocol design as users want to wait as long as possible to vote for the rewards with the highest APY. It makes it hard to know for certain how well a bribe works until the voting period is over, and perhaps some voters wait for just a second too long and miss voting, but otherwise, we have not seen a design solution that remediates this.

Spearbit: Acknowledged.

5.6.8 First `nudge` propose must happen one epoch before tail is set to true

Severity: Informational

Context: Minter.sol#L65-L84

Description: Because you can only `propose` a `nudge` one epoch in advance, the first `propose` call will need to happen on the last epoch in which `tail` is set to `false`

While the transaction simulation will fail for `execute`, the `EpochGovernor.propose` math will make it so that the first proposal will have to be initiated an epoch before in order for it to be executable on the first `tail` epoch

Recommendation: Remind end users about this quirk and perhaps host an event for it

Velodrome: Acknowledged, will update the documentation to make this clearer.

Spearbit: Acknowledged.

5.6.9 Missing emit important events

Severity: Informational

Context: ManagedRewardsFactory.sol#L10 , PairFactory.sol

Description: The contracts that change or create sensible information should emit an event.

Recommendation: Add events to emit for

- `ManagedRewardsFactory`: creating new managed rewards
- `PairFactory`: setVoter, setPauser, setPauseState and setFeeManager

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.10 Marginal rounding errors when using small values

Severity: Informational

Context: Voter.sol#L187

Description: It may be helpful to encourage end users to use BPS or higher denominations for weights when dealing with multiple gauges to keep precision high.

Due to widespread usage of the _vote function throughout the codebase and in forks, it may be best to suggest this in documentation to avoid reverts

Recommendation: It is recommended to validate the weights at a minimum of 1 BPS

Velodrome: I believe this should be handled by the front end. Suggest that minimum increments should be with single basis points? I think at the moment, the front end only lets you select in 1% increments (i.e. 100 BPS). We can add additional documentation to note this though.

Spearbit: Acknowledged.

5.6.11 Prefer to use `nonReentrant` on external functions

Severity: Informational

Context: Voter.sol#L166-L170

Description: it may be best to use `nonReentrant` on the external functions rather than the internal ones. Vote, for example, is not protected because the internal function is.

Recommendation: It is recommended to change the `nonReentrant` modifier from internal `_vote` to external functions `vote` and `poke`.

Velodrome: Fixed in commit 111d83.

Spearbit: Verified.

5.6.12 Redundant variable update

Severity: Informational

Context: Gauge.sol#L180-L211

Description: In `notifyRewardAmount` the variable `lastUpdateTime` is updated twice

Recommendation: It is recommended to remove one update of `lastUpdateTime`

```
 function notifyRewardAmount(uint256 _amount) external nonReentrant {
  ...
- lastUpdateTime = lastTimeRewardApplicable();
  ...
  lastUpdateTime = timestamp;
  ...
 }
```

Velodrome: Fixed in commit a336f7.

Spearbit: Verified.

5.6.13 Turn logic into internal function

Severity: Informational

Context: Gauge.sol#L109-L112, Gauge.sol#L145-148, Gauge.sol#L161-L164

Description: In `Gauge` contract the logic to update `rewardPerTokenStored`,`lastUpdateTime`,`rewards`,`userRewardsPerTokenPai` can be converted to internal function for simplicity

Recommendation: It is recommended to create an internal function and replace this logic

```
 function getReward(address _account) external nonReentrant {
  ...
-  rewardPerTokenStored = rewardPerToken();
-  lastUpdateTime = lastTimeRewardApplicable();
-  rewards[_account] = earned(_account);
-  userRewardPerTokenPaid[_account] = rewardPerTokenStored;
+ _updateRewards(_account);
  ...
 }

+ function _updateRewards(address _account) internal {
+  rewardPerTokenStored = rewardPerToken();
+  lastUpdateTime = lastTimeRewardApplicable();
+  rewards[_account] = earned(_account);
+  userRewardPerTokenPaid[_account] = rewardPerTokenStored;
+}
```

Velodrome: Fixed in commit a336f7.

Spearbit: Verified.

5.6.14 Add extra slippages on client-side when dependent paths are used in `generateZapInParams`

Severity: Informational

Context: Router.sol#L759-L790, Router.sol#L793-L823

Description: `generateZapInParams` is a helper function in `Router` that calculates the parameters for `zapIn`. If there's a duplicate pair in RoutesA and RoutesB, the value calculated here would be off.

For example, The optimal path to swap `dai` into `usdc/velo` pair would likely have dai/eth in both `routesA` and `routesB`. When the user uses this param to call `zapIn`, it executes two swaps:

`dai` -> `eth` -> `usdc`, and `dai` -> `eth` -> `velo`. As the price of dai/eth is changed after the first swap, the second swap would have a slightly bad price. The `zapIn` will likely revert as it does not meet the min token return.

Recommendation: As mentioned by the project team, the front end would add extra slippage when dependent paths are used. Would be good if this is documented.

Velodrome: Will add documentation.

Spearbit: Acknowledged.

5.6.15 Unnecessary `skim` in router

Severity: Informational

Context: Router.sol#L258, Router.sol#L311

Description: The `pair` contract absorbs any extra tokens after `swap`, `mint`, and `burn`. Triggering `Skim` after burn/mint would not return extra tokens.

Recommendation: We can save gas by skipping one external call. We have to be careful when providing liquidity. If the router does not get the optimal token amount on providing liquidity, the user just loses funds. As a result, recommend adjusting the comment in `quoteLiquidity`.

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.16 Overflow is not desired and can lead to loss of funds in `Solidity ^8.0.0`

Severity: Informational

Context: Pair.sol#L235

Description: In `solidity ^8.0`, overflow of `uint` is defaulted to be reverted.

```
//Pair.sol#L235-L239
uint256 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
 if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
     reserve0CumulativeLast += _reserve0 * timeElapsed;
     reserve1CumulativeLast += _reserve1 * timeElapsed;
 }
```

`reserve0CumulativeLast += _reserve0 * timeElapsed;` This calculation will overflow and DOS the pair if `_reserve0` is too large. As a result, the pool should not support high decimals tokens.

Recommendation: High decimals token will break the protocols in many places (e.g. the invariant formula would overflow). This revert will not likely be triggered in any case. Recommend the team just change the misleading comment.

Velodrome: Will update documentation.

Spearbit: Acknowledged.

5.6.17 Unnecessary casting

Severity: Informational

Context: Voter.sol#L148

Description: `_totalWeight` is already declared as uint256

Recommendation: Remove uint256 casting in `_totalWeight`

Velodrome: Fixed in commit 111d83.

Spearbit: Verified.

5.6.18 Refactor retrieve current epoch into library

Severity: Informational

Context: Voter.sol#L81-L83

Description: Could refactor to a library function to retrieve the current epoch

Recommendation: Refactor the logic of retrieving the current epoch into a library function.

Velodrome: Issue address in commit 111d83.

Spearbit: Verified.

5.6.19 Add governor permission to sensible functions

Severity: Informational

Context: VotingEscrow.sol#L217-L222

Description: Some functions that change important variables could add governor permission to enable changes. The function `setManagedState` in VotingEscrow is one that is recommended to add governor permission.

Recommendation: It is recommended to change the code to follow

```
 function setManagedState(uint256 _mTokenId, bool _state) external {
-  require(msg.sender == IVoter(voter).emergencyCouncil(), "VotingEscrow: not emergency council");
+  require(_msgSender() == IVoter(voter).emergencyCouncil() || _msgSender() ==
↪   IVoter(voter).governor(), "VotingEscrow: not emergency council");
   ...
 }
```

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.20 Admin privilege through proposal threshold

Severity: Informational

Context: VeloGovernor.sol#L39-L47

Description: As an Admin Privilege of the Team, the variable `proposalNumerator` could change causing the `proposalThreshold` to be higher than expected. The Team could front-run calls to `propose` and increase the numerator, this could block proposals

Recommendation: As a recommendation, these types of settings should be changed only by multi-sig wallet.

Velodrome: Acknowledged.

Spearbit: Acknowledged.

5.6.21 Simplify check for rounding error

Severity: Informational

Context: Voter.sol#L415, Gauge.sol#L71

Description: The check of rounding error can be simplified. Instead using `A / B > 0` use `A > B`

Recommendation: It is recommended to refactor to

```
//Voter
- if (_claimable > IGauge(_gauge).left() && _claimable / DURATION > 0) {
+ if (_claimable > IGauge(_gauge).left() && _claimable > DURATION) {

//Gauge
- if (_fees0 > IReward(feesVotingReward).left(_token0) && _fees0 / DURATION > 0) {
+ if (_fees0 > IReward(feesVotingReward).left(_token0) && _fees0 > DURATION) {
```

Velodrome: Fixed in commit 111d83.

Spearbit: Verified.

5.6.22 Storage declarations in the middle of the file

Severity: Informational

Context: Voter.sol#L317-L319

Description: If you wish to keep the logic separate, consider creating a separate abstract contract.

Recommendation: It is recommended to create an abstract contract with all storage variables.

Velodrome: Elected to move storage declarations to the top of the file.

Spearbit: Verified.

5.6.23 Inconsistent usage of `_msgSender()`

Severity: Informational

Context: Voter.sol#L75-L78, Voter.sol#L124, VotingEscrow.sol#L63, VotingEscrow.sol#L209, VotingEscrow.sol#L218, VotingEscrow.sol#L234, VotingEscrow.sol#L239, VotingEscrow.sol#L547, VotingEscrow.sol#L256

Description: There are some instances where `msg.sender` is used in contrast with _msgSender() function.

Recommendation: It is recommended to keep the pattern and change `msg.sender` to `_msgSender()`

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.24 Change emergency council should be enabled to `Governor`

Severity: Informational

Context: Voter.sol#L117-L120

Description: Governor may also want to be able to set and change the emergency council, this avoids the potential risk of the council abusing their power

Recommendation: It is recommended to check if the sender is governor

```
 function setEmergencyCouncil(address _council) public {
-  require(_msgSender() == emergencyCouncil, "Voter: not emergency council");
+  require(_msgSender() == emergencyCouncil || _msgSender() == governor, "Voter: not emergency council
↪  or governor");
  emergencyCouncil = _council;
 }
```

Velodrome: List of emergency council permissions

Set emergency council Kill or revive a gauge Set the name / symbol of a pair (NEW) Veto a proposal (unconfirmed) I will consult with the team. I think if the vetoer will be set to emergencyCouncil, it does not make much sense to allow emergencyCouncil to be settable by governor.

Ran this by the team and we'd like to keep things separate:

- Allow veto (Vetoer function) to be called by the team msig: this way the team can veto any governor actions/proposals as a protocol responsible party
- Do not allow the governor to change/set the emergency council msig: this way only the previous emergency council can change the msig of the new council

A bit more reasoning. Our goal is to make Velodrome fully decentralized. The emergency council would eventually be replaced by the governor, meaning that it would make sense to keep the veto right separate from such a change. Naturally, the very next step would be to renounce the Vetoer right from the team.

Spearbit: Acknowledged.

5.6.25 Unnecessary inheritance in `Velo` contract

Severity: Informational

Context: Velo.sol#L9

Description: `Velo` isn't used for governance, therefore it's not necessary to inherit from `ERC20Votes`.

Recommendation: It is recommended to replace `ERC20Votes` with `ERC20Permit`.

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.26 Incorrect value in `Mint` event

Severity: Informational

Context: Minter.sol#L120

Description: In `Minter#update_period` the `Mint` event is emitted with incorrect values.

Recommendation: It is recommended to use updated values

```
- emit Mint(msg.sender, _emission, _totalSupply, _tail);
+ emit Mint(msg.sender, _emission, velo.totalSupply(), _tail);
```

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.27 Do not cache constants

Severity: Informational

Context: Minter.sol#L74

Description: It is not necessary to cache constant variable.

Recommendation: It is recommended to use the constant instead of caching it.

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.28 First week will have no emissions

Severity: Informational

Context: Minter.sol#L54

Description: Cannot call `update_period` on the first week due to setting the current period to this one. Emissions will start at most one week after

Recommendation: It is recommended to set the week in construction to one week before

```
 constructor(...) {
  ...
-  active_period = ((block.timestamp) / WEEK) * WEEK; // allow emissions this coming epoch
+  active_period = (((block.timestamp) / WEEK) * WEEK) - WEEK; // allow emissions this coming epoch
 }
```

Velodrome: I think the logistics of the migration will involve deployment shortly after the epoch flip, with emissions being distributed to gauges for that week (on v1). Migration can then begin, with protocols bribing on the new contracts and votes accumulating on new pools as the week progresses. Then the following week, emissions will be distributed on v2 as normal.

Since `update_period` is callable by anyone, it does not make sense to allow emissions to be distributed immediately as there should be some grace period to allow users to migrate and for votes to populate (e.g. imagine a situation where someone votes for their own pool and then calls `update_period` immediately).

The V1 and V2 will be operating in tandem, and V2 transition will be progressive and will require communication from our side either way. Overall, the bump in emissions on V2, in the beginning, should compensate folks who opt to move their liquidity to V2 gauges in the very first week.

Spearbit: Acknowledged.

5.6.29 Variables can be renamed for better clarity

Severity: Informational

Context: Minter.sol#L23

Description: For a better understanding, some variables could be renamed.

Recommendation: It is recommended to rename the following variable.

```
- uint256 public constant EMISSION = 9_900;
+ uint256 public constant WEEKLY_DECAY = 9_900;
```

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.30 Minter week will eventually shift

Severity: Informational

Context: Minter.sol#L21

Description: The constant `WEEK` is used as the duration of an epoch that resets every Thursday, after 4 years (4 * 365.25 days) the day of the week will eventually shift, not following the Thursday cadence.

Recommendation: This is an informational issue without a fix, that week's shift will eventually happen. Velodrome should be aware of this change

Velodrome: Noted, we will document it somewhere to make it clearer and change references to Thursday in the code.

Spearbit: Acknowledged.

5.6.31 Ownership change will break certain yield farming automations

Severity: Informational

Context: VotingEscrow.sol#L137-L138

Description: Due to the check, any transfer done in the same block as the call to `depositManaged` will revert.

While a sidestep for the mechanic for malicious users was shown, the check will prevent a common use case in Yield Farming: Zapping.

Because of the check, an end user will not be able to zap from their VELO to VE to the Managed Position, which may create a sub-par experience for end users.

This should also create worse UX for Yield Farming Projects as they will have to separate the transfer from the deposit which will cost them more gas and may make their product less capital efficient

Recommendation: Consider the `ownershipChange` lock further and decide if it's worth maintaining or removing

Velodrome: Fixed in commit 02e0bc.

Spearbit: Verified.

5.6.32 Quantitative analysis of `Minter` logic

Severity: Informational

Context: Minter.sol#L32

Description: It will take 110 iterations to go from 15 MLN to the Tail Emission Threshold Minter.sol#L32-L37

```
/// @notice When emissions fall below this amount, begin tail emissions
uint256 public constant TAIL_START = 5_000_000 * 1e18;
/// @notice Tail emissions rate in basis points
uint256 public tailEmissionRate = 30;
/// @notice Starting weekly emission of 15M VELO (VELO has 18 decimals)
uint256 public weekly = 15_000_000 * 1e18;
```

```
## Python allows decimal Math, wlog, just take mantissa
INITIAL = 15 * 10 ** 6
TAIL = 5 * 10 ** 6
MULTIPLIER_BPS = 99_00
MAX_BPS = 10_000
value = INITIAL
i = 0
min_emitted = 0
while (value > TAIL):
  i+= 1
  min_emitted += value
  value = value * MULTIPLIER_BPS / MAX_BPS

i
110
value
4965496.324815211
min_emitted
1003450367.5184793

## If nobody ever bridged, this would be emissions at tail
min_emitted * 30 / 10_000
3010351.1025554384
```

Tail emissions are most likely going to be a discrete step down in emissions

```
>>> min_emitted
1003450367.5184793

V1_CIRC = 150 * 10 ** 6
ranges = range(V1_CIRC // 10, V1_CIRC, V1_CIRC // 10)

for val in ranges:
  print((min_emitted + val) * 30 / 10_000)

3055351.1025554384
3100351.1025554384
3145351.1025554384
3190351.1025554384
3235351.1025554384
3280351.1025554384
3325351.1025554384
3370351.1025554384
3415351.1025554384
```

The last value before the tail will be most likely around 1 Million fewer tokens minted per period. Maximum Mintable Value is slightly above Tail, with Absolute Max being way above Tail

```
## Max Supply
>>> 1000 * 10 ** 6
1000000000
>>> min_emitted = 1003450367.5184793
>>> max_circ = 1000 * 10 ** 6 + min_emitted
>>> max_mint = max_circ * 30 / 10_000

## If we assume min_emitted + 1 Billion Velo V1 Sinked
>>> max_mint
6010351.102555438

## If we assume nudge to 100 BPS
>>> abs_max_mint = max_circ * 100 / 10_000
>>> abs_max_mint
20034503.675184794
```

Recommendation: Consider if these are the intended consequences.

Velodrome: QA 1: 110 weeks look alright.

QA 2:

```
CUR_EPOCH = 38
TAIL_EPOCH = 110

CUR_SUPPLY = 841884804.4495
CUR_VE_SUPPLY = 706050740

next_lp, next_rebase = 10238318.925 * 0.99, CUR_VE_SUPPLY * 0.2/52
total_supply = CUR_SUPPLY
ve_supply = CUR_VE_SUPPLY

for week in range(CUR_EPOCH, TAIL_EPOCH):
    total_supply += next_lp + next_rebase
    # team
    total_supply += 0.03 * (next_lp + next_rebase)
    ve_supply = 0.8 * total_supply
    next_lp, next_rebase = next_lp * 0.99, ve_supply * 0.2/52
```

(Assuming 20% rebase APY and 80% lock rate)

Results:

```
>>> total_supply
1668569723.3304946
>>> total_supply * 30 / 10_000
5005709.169991484
```

QA 3: Tbh I can't see anything wrong here, even if the governance of velofed decides tailEmissionRate should be up to the maximum it's still working as intended. From the economical design looks like we chose to multiply tailEmissionRate with totalsupply instead of scaling a fixed value like 5M, so the amount minted is going to be a divergent series even with a fixed tailEmissionRate.

Spearbit: Acknowledged.

5.6.33 Optimism's block production may change in the future

Severity: Informational

Context: GovernorSimpleVotes.sol#L22, VeloGovernor.sol#L31

Description: In contrast to `OZ's Governor` which uses timestamp, the `GovernorSimpleVotes` contract uses block number, because of OP potentially changing block frequency in the future, given Bedrocks update to block.timestamp, it may be desirable to refactor back to the OZ implementation. And `VeloGorvernor` assumes 2 blocks every second.

In OP's docs says block.number is not a reliable timing reference: community.optimism.io/docs/developers/build/differences/#blo numbers-and-timestamps

It's also dangerous to use block.number at the time cause it will probably mean a different thing in pre- and post-bedrock upgrades.

Recommendation: The recommendation is to refactor back to OZ implementation.

Velodrome: Given OpenZeppelin's support for a more flexible Governor that can allow contracts to select a clock of their liking, we have elected to refactor back to a timestamp based system while choosing timestamps as the default clock for VotingEscrow. Refactor towards timestamp based Governors in this PR: commit 08c2bc.

Spearbit: Verified.

5.6.34 Remove unnecessary check

Severity: Informational

Context: GovernorSimple.sol#L266-L267

Description: These checks are unnecessary because it already checks if targets and calldata lengths are equal to 1.

Recommendation: Remove the checks

```
- require(targets.length == values.length, "GovernorSimple: invalid proposal length");
- require(targets.length == calldatas.length, "GovernorSimple: invalid proposal length");
```

Velodrome: Acknowledged and will fix.

Spearbit: Acknowledged.

5.6.35 Event is missing indexed fields

Severity: Informational

Context: Pair.sol#L70-L73, Pair.sol#L81-L82, IFactoryRegistry.sol#L5-L7, IGauge.sol#L4-L8, IMinter.sol#L4, IMinter.sol#L6, IPairFactory.sol#L4-L5, IReward.sol#L4-L7, IRewardDistributor.sol#L4-L5, ISinkManager.sol#L7-L8, ISinkManager.sol#L16, IVoter.sol#L13-L18, IVoterEscrow.sol#L42, IVoterEscrow.sol#L50-L52, IVoterEscrow.sol#L74, SinkConverter.sol#L21

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Recommendation: Consider ensuring that all events are correctly indexed to the protocols needs.

Velodrome: Acknowledged, we will make changes accordingly.

Spearbit: Acknowledged.

5.6.36 Missing checks for address(0) when assigning values to address state variables

Severity: Informational

Context: FactoryRegistry.sol#L26-L28, RewardsDistributor.sol#L39-L40, RewardsDistributor.sol#L308, Velo.sol#L22, Velo.sol#L28, VeloGovernor.sol#L36, Voter.sol#L101, Voter.sol#L107, Voter.sol#L113, Voter.sol#L119, VotingEscrow.sol#L64, VotingEscrow.sol#L66, VotingEscrow.sol#L235, VotingEscrow.sol#L240, VotingEscrow.sol#L1114, PairFactory.sol#L54, PairFactory.sol#L64-L65, PairFactory.sol#L85, PairFactory.sol#L95, PairFactory.sol#L87, PairFactory.sol#L12, SinkManager.sol#L134, PairFactory.sol#L64-L66

Description: Lack of zero-address validation on address parameters may lead to transaction reverts, waste gas, require resubmission of transactions and may even force contract redeployments in certain cases within the protocol.

Recommendation: Consider adding explicit zero-address validation on input parameters of address type.

Velodrome: Acknowledged.

Spearbit: Acknowledged.

5.6.37 Incorrect comment

Severity: Informational

Context: Mentioned in Recommendation

Description: There are a few mistakes in the comments that can be corrected in the codebase.

Recommendation:

- Router.sol#L148: Invalid comment as it does not create a new pair if it doesn't exist yet within this function.

```
- // create the pair if it doesn't exist yet
  address _pair = IPairFactory(_factory).getPair(tokenA, tokenB, stable);
  (uint256 reserveA, uint256 reserveB) = (0, 0);
```

- Router.sol#L179: Invalid comment as it does not create a new pair if it doesn't exist yet within this function.

```
  require(amountADesired >= amountAMin);
  require(amountBDesired >= amountBMin);
- // create the pair if it doesn't exist yet
  address _pair = IPairFactory(defaultFactory).getPair(tokenA, tokenB, stable);
```

Velodrome: Fixed in commit eb7f0ed.

Spearbit: Verified.

5.6.38 Discrepancies between specification and implementation

Severity: Informational

Context: Router.sol#L579, Router.sol#L718

Description: Instance 1 - Zapping

The specification mentioned that it supports zapping into a pool from **any** token. Following is the extract

- Swapping and lp depositing/withdrawing of fee-on-transfer tokens.
- Zapping in and out of a pool from any token (i.e. A->(B,C) or (B,C) -> A). A can be the same as B or C.
- Zapping and staking into a pool from any token.

However, the `zapIn` and `zapOut` functions utilize the internal `_swap` function that does not support fee-on-transfer tokens.

Recommendation: Consider updating the specification to explicitly state that zapping is not supported for fee-on-transfer tokens. Additionally, consider adding comments to the zapping function to highlight this limitation.

Velodrome: Fixed in commit eb7f0e.

Spearbit: Verified.

5.6.39 Early exit for `withdrawManaged` function

Severity: Informational

Context: VotingEscrow.sol#L165

Description: If `_mTokenId` is zero, there is no point executing the rest of the code within the `VotingEscrow.withdrawManaged` function.

Recommendation: For defense in depth, consider reverting and exiting the function immediately if `_mTokenId` is zero to avoid any potential edge cases when executing the rest of the code.

Velodrome: Fixed in commit 827917.

Spearbit: Verified.

6 Appendix

6.1 Appendix: Summary

The findings in this section correspond to the post-engagement review conducted over a split period of two separate weeks, the week of May 22nd to May 26th and the week of June 12th to June 16th.

The commit hash at e5635f was reviewed during the first period. Fixes and other modifications were implemented by the Velodrome team in between the first and second period. During the second period, the following commit was used 274c77.

6.2 High Risk

6.2.1 DOS attack at future facilitator contract and stop `SinkManager.convertVe`

Severity: High Risk

Context: SinkManager.sol#L123

Description: As noted in *"DOS attack by delegating tokens at MAX_DELEGATES = 1024"*, the old `votingEscrow` has a gas concern, i.e., the gas cost of transfer/ burn will increase when an address holds multiple NFT tokens. The concern becomes more serious when the protocol is deployed on Optimism, where the gas limit is smaller than other L2 chains. If an address is being attacked and holds max NFT tokens (1024), the user can not withdraw funds due to the gas limit.

To mitigate the potential DOS attack where the attack DOS the v1's `votingEscrow` and stop `sinkManager` from receiving tokens, the `sinkManager` utilize a facilitator contract. When the sinkManager needs to receive the votingEscrow NFT, it creates a new contract specifically for this purpose. Since the contract is newly created, it does not contain any tokens, making it more gas-efficient to receive the token through the facilitator contract.

However, the attacker can DOS attack the contract by sending NFT tokens to a future facilitator.

salted-contract-creations-create2

> When creating a contract, the address of the contract is computed from the address of the creating contract and a counter that is increased with each contract creation.

The exploit scenario would be: At the time the `sinkManager` is deployed and zero facilitator is created. The attacker can calculate the address of all future facilitators by computing `sha3(rlp.encode([normalize_address(sender), nonce]))[12:]` The attacker can compute the 10-th facilitator's address and sends 1024 NFT tokens to the address. The `sinkManager` will function normally nine times. Though, when the 10th user wants to convert the token, the `sinkManager` deployed the 10th facilitator address. Since the 10th facilitator already has 1024 NFT positions, it can not receive any tokens. The transaction will revert and the sinkManager will be stuck in the current state.

Recommendation: Recommend to use `cloneDeterministic` (Clones.sol#L45-L56) with an unpredictable salt.

```
uint256 private counter;
// ...
/// @inheritdoc ISinkManager
function convertVe(uint256 tokenId) external nonReentrant returns (uint256 tokenIdV2) {
{
// ...
  // Create contract to facilitate the merge
  SinkManagerFacilitator facilitator =
  ↪  SinkManagerFacilitator(Clones.cloneDeterministic(facilitatorImplementation,
  ↪  sha3(abi.encodedcounter, blockhash(block.number - 1))));
  counter++;
 // ...
}
```

Two behaviors the fix try to achieve:

1. Put some randomness to the facilitator's address so that the attacker can't predict and attack it.

2. The `sinkManager` should not be stopped even if one address failed to receive tokens.

Velodrome: Fixed in commit 7b35b2. In addition to `blockhash(block.number-1)`, additional user-provided salt is provided to further increase the cost of griefing attacks.

Spearbit: Verified.

6.2.2 `RewardDistributor` caching `totalSupply` leading to incorrect reward calculation

Severity: High Risk

Context: RewardsDistributor.sol#L136-L159

Description: `RewardDistributor` distributes newly minted VELO tokens to users who locks the tokens in `VotingEscrow`. Since the calculation of past supply is costly, the `rewardDistributor` cache the supply value in `uint256[1000000000000000] public veSupply`. The `RewardDistributor._checkpointTotalSupply` function would iterate from the last updated time util the latest epoch time, fetches totalSupply from `votingEscrow`, and store it.

Assume the following scenario when a transaction is executed at the beginning of an epoch.

1. The `totalSupply` is X.
2. The user calls `checkpointTotalSupply`. The `rewardDistributor` save the `totalSupply = X`.
3. The user creates a lock with 2X the amount of tokens. The user has `balance = 2X` and the `totalSupply` becomes 3X.
4. Fast forward to when the reward is distributed. The user claims the tokens, reward is calculated by `total reward * balance / supply` and user gets 2x of the total rewards.

Recommendation: This issue shares a lot of similarities to *"Reward calculates earned incorrectly on each epoch boundary"* at a lower severity.

The quick fix would be to stop `rewardDistributor` caching `totalSupply` when it can still increase.

```
    function _checkpointTotalSupply() internal {
        address _ve = ve;
        uint256 t = timeCursor;
        uint256 roundedTimestamp = (block.timestamp / WEEK) * WEEK;
        IVotingEscrow(_ve).checkpoint();

        for (uint256 i = 0; i < 20; i++) {
-           if (t > roundedTimestamp) {
+           if (t >= roundedTimestamp) {
                break;
            } else {
                // fetch last global checkpoint prior to time t
                uint256 epoch = _findTimestampEpoch(t);
                IVotingEscrow.GlobalPoint memory pt = IVotingEscrow(_ve).pointHistory(epoch);
                int128 dt = 0;
                if (t > pt.ts) {
                    dt = int128(int256(t - pt.ts));
                }
                // walk forward voting power to time t
                veSupply[t] = uint256(int256(max(pt.bias - pt.slope * dt, 0))) +
                ↪  pt.permanentLockBalance;
            }
            t += WEEK;
        }
        timeCursor = t;
    }
```

Given similar issues occur multiple times in the code base and the nuance of this issue can be. We recommend not caching the totalSupply in `rewardDistributor` and fetching `totalSupply` from the votingEscrow every time is needed.

Velodrome: Fixed in commit 2d7f02. RewardDistributor now fetches the 1 second before the epoch flip of`totalSupply` and `balanceOf` from `votingEscrow`.

Spearbit: Verified.

6.3 Medium Risk

6.3.1 Lack of slippage control during compounding

Severity: Medium Risk

Context: AutoCompounder.sol#L78, AutoCompounder.sol#L91

Description: When swapping the reward tokens to VELO tokens during compounding, the slippage control is disabled by configuring the `amountOutMin` to zero. This can potentially expose the swap/trade to sandwich attacks and MEV (Miner Extractable Value) attacks, resulting in a suboptimal amount of VELO tokens received from the swap/trade.

```
router.swapExactTokensForTokens(
    balance,
    0, // amountOutMin
    routes,
    address(this),
    block.timestamp
);
```

Recommendation: Consider implementing some form of slippage control to reduce the risks associated with sandwich attacks and MEV attacks. Since the `claimBribesAndCompound` and `claimFeesAndCompound` functions are permissionless, the slippage parameters should be dynamically computed before the trade. Otherwise, malicious users could set an excessively high slippage tolerance enabling large price deviations. This, in combination with a sandwich attack to extract the maximum value from the trade.

One possible solution is to dynamically compute the minimum amount of VELO tokens to be received after the trade based on the maximum allowable slippage percentage (e.g. 5%) and the exchange rate (Source Token <> VELO) from a source that cannot be manipulated (e.g. Chainlink, Custom TWAP).

Alternatively, consider restricting access to these functions to only certain actors who can be trusted to define an appropriate slippage parameter where possible.

Velodrome: See commit b4283f for the proposed fix.

Another thing to note is that Optimism does not currently enable MEV / sandwiching due to the private mempool so the only probability in this MEV happening is in a protocol change.

Spearbit: Fixed.

6.3.2 `ALLOWED_CALLER` can steal all rewards from `AutoCompounder` using a fake factory in the route.

Severity: Medium Risk

Context: AutoCompounder.sol#L187

Description: `AutoCompounder` allows address with `ALLOWED_CALLER` role to trigger `swapTokenToVELOAndCompound`. The function sells the specified tokens to `VELO`.

Since the Velo router supports multiple factories. An attacker can deploy a fake factory with a backdoor. By routing the swaps through the backdoor factory the attacker can steal all reward tokens in the `AutoCompounder` contract.

Recommendation:: As mentioned by the Velodrome team, we can only allow certain factories to be in the route.

One potential solution could be to check the factory in the routes against the registry to see if it is approved.

Besides the fix, we should acknowledge the risk and pay more attention to permission management. The `ALLOWED_CALLER` can always steal the rewards. As noted in this issue: *"Lack of slippage control during compounding"*, an attacker can steal the tokens by sandwiching the trades. This is the current limitation of DEFI. There's no easy way to do it without an oracle.

Velodrome: Fixed code in commit 24e50b. We had originally implemented the fix and then determined the bigger risk for a user would be calling the router from a frontend that can pass in any arbitrary factory. This would be a risk as a compromised website could correctly interact with the router by using a fake factory and act maliciously. With commit 24e50b. we ensure that any interaction with the router is done with a PoolFactory approved by our registry. We also made a change to the registry where once a PoolFactory is approved, it will always appear as a registered PoolFactory.

Spearbit: Verified.

6.3.3 `depositManaged` can be used by locks to receive unvested VELO rebase rewards

Severity: Medium Risk

Context: RewardsDistributor.sol#L134-L143

Description: Velo offer rebase emissions to all Lockers. These are meant to be `depositFor` into an existing lock, and directly transferred if the lock just expired. The check is the following:

```
if (_timestamp > _locked.end && !_locked.isPermanent) {
```

By calling `depositManaged` we can get the check to pass for a Lock that is not expired, allowing us to receive Unvested Velo (we could sell unfairly for example).

Due to how `depositManaged` and `withdrawManaged` work, the attacker would be able to perform this every other week (1 week cooldown, 1 week execution).

Because of how the fact that `managedRewards` are delayed by a week the attacker will not lose any noticeable amount of rewards, meaning that most users would rationally opt-into performing this operation to gain an unfair advantage, or to sell their rewards each week while other Lockers are unable or unwilling to perform this operation.

The following POC will show an increase in VELO balance for the `tokenId2` owner in spite of the fact that the lock is not expired

```
Logs:
  Epoch 1
  Token Locked after  56039811453980167852
  Token2 Locked after  -100000000000000000000000 // Negative because we have `depositManaged`
  User Bal after  56039811453980167852 // We received the token directly, unvested
```

```
function testInstantClaimViaManaged() public {
    // Proof that if we depositManaged, we can get our rebase rewards instantly
    // Instead of having to vest them via the lock
    skipToNextEpoch(1 days);
    minter.updatePeriod();

    console2.log("Epoch 1");

    VELO.approve(address(escrow), TOKEN_1M * 2);
    uint256 tokenId = escrow.createLock(TOKEN_1M, MAXTIME);
    uint256 tokenId2 = escrow.createLock(TOKEN_1M, MAXTIME);
    uint256 mTokenId = escrow.createManagedLockFor(address(this));


    skipToNextEpoch(1 hours + 1);
    minter.updatePeriod();
```

```
    skipToNextEpoch(1 hours + 1);
    minter.updatePeriod();

    // Now we claim for 1, showing that they incease locked
    int128 initialToken1 = escrow.locked(tokenId).amount;
    distributor.claim(tokenId); // Claimed from previous epoch
    console2.log("Token Locked after ", escrow.locked(tokenId).amount - initialToken1);

    // For 2, we deposit managed, then claim, showing we get tokens unlocked
    uint256 initialBal = VELO.balanceOf(address(this));
    int128 initialToken2 = escrow.locked(tokenId2).amount;
    voter.depositManaged(tokenId2, mTokenId);
    distributor.claim(tokenId2); // Claimed from previous epoch

    console2.log("Token2 Locked after ", escrow.locked(tokenId2).amount - initialToken2);
    console2.log("User Bal after ", VELO.balanceOf(address(this)) - initialBal);
}
```

Recommendation: At this time, it seems like the best suggestion would be to check if the token is managed, and if it is, to `depositFor` the managed token.

Velodrome: This finding is valid and a user can in fact circumvent the rebase to their account instead of their tokenId. There are a couple key things to note here:

- `depositFor()` into the (m)veNFT would distribute the rebase to all lockers within the (m)veNFT.
- `distributor.claim()` is publicly callable on the behalf of any tokenId.

Given these conditions, a user may unknowingly have a rebase to claim but deposit into the (m)veNFT. If we went with the proposed solution of `depositFor()` into (m)veNFT, a malicious caller could see this pending claim and have it distributed to everyone within the (m)veNFT.

I believe the best solution would integrate `distributor.claim()` within `depositManaged()` so that the claim is automatically done and given to the rightful owner.

Spearbit: Great points. A check has been added that causes a revert on claiming if the Lock is in a Managed Position that also may be sufficient. By not allowing the tokens to be distributed, they will be distributed exclusively to the lock once it's withdrawn from the managed position. This should ensure that those rebase rewards are at least unlocked at the time in which the user lock expires, which seems more in line with the design of not allowing rebase rewards to be liquid immediately.

Velodrome: Right, the fix you mention is within commit 0323be.

Spearbit: Fixed.

6.4 Low Risk

6.4.1 Unnecessary slippage loss due to `AutoCompounder` selling `VELO`

Severity: Low Risk

Context: AutoCompounder.sol#L78-L85 AutoCompounder.sol#L91-L98

Description: `AutoCompounder` allows every address to help claim the rewards and compound to the locked VELO position. The `AutoCompounder` will sell `_tokensToSwap` into `VELO`. By setting `VELO` as `_tokensToSwap`, the `AutoCompounder` would do unnecessary swaps that lead to unnecessary slippage loss.

Recommendation: Checks `_tokensToSwap != address(VELO)` .

Velodrome: Fixed in commit ffb016.

Spearbit: Fixed.

6.4.2 `epochVoteStart` function calls the wrong library method

Severity: Low Risk

Context: Voter.sol#L111

Description: The `epochVoteStart` function calls the `VelodromeTimeLibrary.epochStart` function instead of the `VelodromeTimeLibrary.epochVoteStart` function. Thus, the `Voter.epochVoteStart` function returns a voting start time without factoring in the one-hour distribution window, which might cause issues for users and developers relying on this information.

Recommendation: Consider making the following change to the `Voter.epochVoteStart` function

```
 function epochVoteStart(uint256 _timestamp) external pure returns (uint256) {
- return VelodromeTimeLibrary.epochStart(_timestamp);
+ return VelodromeTimeLibrary.epochVoteStart(_timestamp);
```

Velodrome: Fixed in commit f7eb2f.

Spearbit: Verified.

6.4.3 Managed NFT can vote more than once per epoch under certain circumstances

Severity: Low Risk

Context: Voter.sol#L286

Description: The owner of the managed NFT could break the invariant that an NFT can only vote once per epoch

Assume Bob owns the following two (2) managed NFTs:

- Managed veNFT (called $mNFT_a$) with one (1) locked NFT (called $lNFT_a$)
- Managed veNFT (called $mNFT_b$) with one (1) locked NFT (called $lNFT_b$)
- The balance of $lNFT_a$ and $lNFT_b$ is the same

Bob voted on $pool_x$ with $mNFT_a$ and $mNFT_b$ on the first hour of the epoch

At the last two hours of the voting windows of the current epoch, Bob changed his mind and decided to vote on the $pool_y$.

Under normal circumstances, the `onlyNewEpoch` modifier will prevent $mNFT_a$ and $mNFT_b$ from triggering the `Voter.vote` function because these two veNFTs have already voted in the current epoch and their `lastVoted` is set to a timestamp within the current epoch.

However, it is possible for Bob to bypass this control. Bob could call `Voter.withdrawManaged` function to withdraw $lNFT_a$ and $lNFT_b$ from $mNFT_a$ and $mNFT_b$ respectively. Since the weight becomes zero, the `lastVoted` for both $mNFT_a$ and $mNFT_b$ will be cleared. As a result, they will be allowed to re-vote in the current epoch.

Bob will call `Voter.depositManaged` to deposit $lNFT_b$ into *mNFTa* and $lNFT_a$ into $mNFT_b$ respectively to increase the weight of the managed NFTs.

Bob then calls `Voter.vote` with *mNFTa* and $mNFT_b$ to vote on $pool_y$. Since the `lastVoted` is empty (cleared earlier), the `onlyNewEpoch` modifier will not revert the transaction.

Understood that the team that without clearing the `lastVoted`, it would lead to another potential issue where a new managed NFT could potentially be made useless temporarily for an epoch. Given the managed NFT grant significant power to the owner, the team intended to restrict access to the managed NFTs and manage abuse by utilizing the emergency council/governor to deactivate non-compliant managed NFTs, thus mitigating the risks of this issue.

Recommendation: To prevent any managed NFT owner from abusing this issue, consider disallowing re-voting in the current epoch once the last NFT has been withdrawn from a managed NFT.

Velodrome: This potential interaction was built from commit 6b95f9 and is going to remain in the codebase. We acknowledge there is a risk of vote signaling abuse by changing the vote last second. However, there is a greater

risk to the managed veNFT holder losing their ability to vote if lockers decide to withdraw from the (m)veNFT. In signaling abuse, trust is given to the (m)veNFT voter, the same user who has the power to vote. In disallowing re-voting, trust is given to the lockers, of whom the (m)veNFT voter has no control over. We understand the additional economics of (m)veNFT vote changes and intend to create (m)veNFT on a partner-by-partner basis. If this voting change is abused by a partner the team is able to disable the (m)veNFT from voting again.

Spearbit: Acknowledged.

6.4.4 Invalid route is returned if `token` does not have a trading pool

Severity: Low Risk

Context: CompoundOptimizer.sol#L79

Description: Assume that someone called the `getOptimalTokenToVeloRoute` function with a token called *T* that does not have a trading pool within Velodrome.

While looping through all the ten (two) routes pre-defined in the constructor at Line 94 below, since the trading pool with *T* does not exist, it will keep skipping to the next route until the loop ends. As such, the `index` remains uninitialized at the end, meaning it holds the default value of zero.

In Lines 110 to 112, it will conclude that the optimal route is as follows:

```
routes[0] = routesTokenToVelo[index][0] = routesTokenToVelo[0][0] = address(0) <> USDC
routes[1] = routesTokenToVelo[index][1] = routesTokenToVelo[0][1] = USDC <> VELO
routes[0].from = token = T
routes = T <> USDC <> VELO
```

As a result, the `getOptimalTokenToVeloRoute` function returns an invalid route.

```
function getOptimalTokenToVeloRoute(
    address token,
    uint256 amountIn
) external view returns (IRouter.Route[] memory) {
    // Get best route from multi-route paths
    uint256 index;
    uint256 optimalAmountOut;
    IRouter.Route[] memory routes = new IRouter.Route[](2);
    uint256[] memory amountsOut;
    // loop through multi-route paths
    for (uint256 i = 0; i < 10; i++) {
        routes[0] = routesTokenToVelo[i][0];
        // Go to next route if a trading pool does not exist
        if (IPoolFactory(routes[0].factory).getPair(token, routes[0].to, routes[0].stable) ==
        ↪  address(0)) continue;
        routes[1] = routesTokenToVelo[i][1];
        // Set the from token as storage does not have an address set
        routes[0].from = token;
        amountsOut = router.getAmountsOut(amountIn, routes);
        // amountOut is in the third index - 0 is amountIn and 1 is the first route output
        uint256 amountOut = amountsOut[2];
        if (amountOut > optimalAmountOut) {
            // store the index and amount of the optimal amount out
            optimalAmountOut = amountOut;
            index = i;
        }
    }
    // use the optimal route determined from the loop
    routes[0] = routesTokenToVelo[index][0];
    routes[1] = routesTokenToVelo[index][1];
    routes[0].from = token;
    // Get amountOut from a direct route to VELO
    IRouter.Route[] memory route = new IRouter.Route[](1);
```

```
    route[0] = IRouter.Route(token, velo, false, factory);
    amountsOut = router.getAmountsOut(amountIn, route);
    // compare output and return the best result
    return amountsOut[1] > optimalAmountOut ? route : routes;
}
```

Recommendation: If none of the routes support the `token`, consider reverting the transaction instead of returning a route built upon an uninitialized `index`.

```
+ bool routeFound = false;

 for (uint256 i = 0; i < 10; i++) {
    routes[0] = routesTokenToVelo[i][0];

    // Go to next route if a trading pool does not exist
-    if (IPoolFactory(routes[0].factory).getPair(token, routes[0].to, routes[0].stable) == address(0))
↪   continue;
+    if (IPoolFactory(routes[0].factory).getPair(token, routes[0].to, routes[0].stable) == address(0)) {
+        continue;
+    } else {
+        routeFound = true;
+    }
..SNIP..
}

+ if (!routeFound) revert NoRouteFound();
```

Velodrome: Fixed in commit 367ecf.

Spearbit: Verified.

6.4.5 `SafeApprove` is not used in `AutoCompounder`

Severity: Low Risk

Context: AutoCompounder.sol#L186 AutoCompounder.sol#L109

Description: `safeApprove` is not used in `AutoCompounder`. Tokens that do not follow standard `ERC20` will be locked in the contract.

Recommendation: Use `safeApprove` and clear the allowance before calling token contracts.

```
IERC20(token).safeApprove(address(router), 0);
IERC20(token).safeApprove(address(router), balance);
```

Velodrome: Fixed in commit 10ae5c.

Spearbit: Verified.

6.4.6 `balanceOfNFT` can be made to return non-zero value via `split` and `merge`

Severity: Low Risk

Context: VotingEscrow.sol#L1059-L1060

Description: Ownership Change Sidestep via Split. Splitting allows to change the ID, and have it work. This allows to sidestep this check in VotingEscrow.sol#L1052-L1055

Meaning you can always have a non-zero balance although it requires performing some work. This could be used by integrators as a way to accurately track their own voting power.

Recommendation: At this time, no specific vulnerability was found besides the ability to sidestep this view function, so no action seems to be required beside flagging this to future integrators.

Velodrome: A couple things here:

- In `merge`, one veNFT is burned and the other increases balance. In `split`, two fresh veNFTs are minted to the owner of the split veNFT, and the split veNFT is burned.
- Ownership change tracking are solely done to protect against flash-voting from `transferFrom()`.

It's worth acknowledging that yes, from these functions the locked balance changes. However, I do not see a vulnerability from these functions.

Spearbit: By performing a split and merge you'd be able to vote in the same block, however, you'd be doing so by breaking the lock and using another tokenId.

Something important to consider around flashloans with splitting, is that you could split the token to leave a dust amount (but maintain the tokenId) and you'd be able to vote because the newly create tokenId would not have ownershipChange set

We weren't able to use this to bypass anything meaningful besides being able to use _delegate or balanceOf in the same tx / block as the transferFrom

6.4.7 `delegateBySig` can use malleable signatures

Severity: Low Risk

Context: VotingEscrow.sol#L1202-L1203

Description: Because the function `delegateBySig` uses `ecrecover` and doesn't check for the value of the signature, other signatures, that have higher numerical values, which map to the same signature, could be used. Because the code uses `nonces` only one signature could be used per `nonce`.

Recommendation: Consider using `ECDSA` by Open Zeppelin, or adding the check they use here.

Velodrome: Fixed in commit ebe3af.

Spearbit: Fixed.

6.4.8 Slightly Reduced Voting Power due to Rounding Error

Severity: Low Risk

Context: BalanceLogicLibrary.sol#L99-L100

Description: Because of rounding errors, a fully locked NFT will incur a slight loss of Vote Weight (around 27 BPS).

```
[PASS] testCompareYieldOne() (gas: 4245851)
Logs:
  distributor.claimable(tokenId) 0
  locked.amount 100000000000000000000000
  block.timsestamp 1814399
  block.timsestamp 1900800
  Epoch 2
  distributor.claimable(tokenId) 0
  locked.amount 100000000000000000000000
  escrow.userPointHistory(tokenId, 1) 0
  escrow.userPointHistory(tokenId, 1) 1814399
  escrow.userPointHistory(tokenId, 1) BIAS 997260281900050656907546
  escrow.userPointHistory(tokenId, tokenId2) 1814399
  escrow.userPointHistory(tokenId, tokenId2) BIAS 997260281900050656907546
  userPoint.ts 1814399
  getCursorTs(tokenId) 1814400
  userPoint.ts 1814399
  epochStart(tokenId) 1814400
```

```
  userPoint.ts 1814399
  ve.balanceOfNFTAt(tokenId, getCursorTs(tokenId) - 1) 997260281900050656907546
```

```
function getCursorTs(uint256 tokenId) internal returns(uint256) {
            IVotingEscrow.UserPoint memory userPoint = escrow.userPointHistory(tokenId, 1);
            console2.log("userPoint.ts", userPoint.ts);
            uint256 weekCursor = ((userPoint.ts + WEEK - 1) / WEEK) * WEEK;
            uint256 weekCursorStart = weekCursor;
            return weekCursorStart;
    }

    function epochStart(uint256 timestamp) internal pure returns (uint256) {
        unchecked {
            return timestamp - (timestamp % WEEK);
        }
    }

    function testCompareYieldOne() public {
        skipToNextEpoch(1 days);
        // Epoch 1

        skipToNextEpoch(-1); // last second
        VELO.approve(address(escrow), TOKEN_1M * 2);
        uint256 tokenId = escrow.createLock(TOKEN_1M, MAXTIME);
        uint256 tokenId2 = escrow.createLock(TOKEN_1M, 4 * 365 * 86400);
        uint256 mTokenId = escrow.createManagedLockFor(address(this));

        console2.log("distributor.claimable(tokenId)", distributor.claimable(tokenId));
        console2.log("locked.amount", escrow.locked(tokenId).amount);

        console2.log("block.timsestamp", block.timestamp);

        minter.updatePeriod(); // Update for 1
        skipToNextEpoch(1 days); // Go next epoch
        minter.updatePeriod(); // and update 2

        console2.log("block.timsestamp", block.timestamp);

        console2.log("Epoch 2");

        //@audit here we have claimable for tokenId and mTokenId
        IVotingEscrow.LockedBalance memory locked = escrow.locked(tokenId);
        console2.log("distributor.claimable(tokenId)", distributor.claimable(tokenId));
        console2.log("locked.amount", escrow.locked(tokenId).amount);

        console2.log("escrow.userPointHistory(tokenId, 1)", escrow.userPointHistory(tokenId, 0).ts);
        console2.log("escrow.userPointHistory(tokenId, 1)", escrow.userPointHistory(tokenId, 1).ts);
        console2.log("escrow.userPointHistory(tokenId, 1) BIAS", escrow.userPointHistory(tokenId,
        ↪  1).bias);
        console2.log("escrow.userPointHistory(tokenId, tokenId2)", escrow.userPointHistory(tokenId2,
        ↪  1).ts);
        console2.log("escrow.userPointHistory(tokenId, tokenId2) BIAS",
        ↪  escrow.userPointHistory(tokenId2, 1).bias);

        console2.log("getCursorTs(tokenId)", getCursorTs(tokenId));
        console2.log("epochStart(tokenId)", epochStart(getCursorTs(tokenId)));

        console2.log("ve.balanceOfNFTAt(tokenId, getCursorTs(tokenId) - 1)",
        ↪  escrow.balanceOfNFTAt(tokenId, getCursorTs(tokenId) - 1));
    }
```

Recommendation: A nofix seems acceptable given the ability for end users to use `permanent locks`. Since these types of locks offer 100% of the voting power, they may be the preferred option for end users.

Velodrome: The team is aware of this and has kept it intentionally to encourage permanent locks.

Spearbit: Acknowledged.

6.4.9 Some setters cannot be changed by governance

Severity: Low Risk

Context: Voter.sol#L151-L155

Description: It was found that some setters, related to `emergencyCouncil` and `Team` can only be called by the current role owner. It may be best to allow `Governance` to also be able to call such setters as a way to allow it to override or replace a misaligned team.

The Emergency Council can kill gauges, preventing those gauges from receiving emissions.

Voter.sol#L151-L155.

```
function setEmergencyCouncil(address _council) public {
    if (_msgSender() != emergencyCouncil) revert NotEmergencyCouncil();
    if (_council == address(0)) revert ZeroAddress();
    emergencyCouncil = _council;
}
```

The team can simply change the ArtProxy which is a cosmetic aspect of Voting Escrow.

VotingEscrow.sol#L241-L245

```
function setTeam(address _team) external {
    if (_msgSender() != team) revert NotTeam();
    if (_team == address(0)) revert ZeroAddress();
    team = _team;
}
```

Recommendation: Consider allowing Governance to change the addresses with the roles.

Velodrome: This design is intentional to prevent a governance takeover. There is always the ability to give governance the team role once governance has been properly established.

Spearbit: Acknowledged.

6.4.10 Rebase Rewards distribution is shifted by one week, allowing new depositors to receive unfair yield initially (which they'll give back after they withdraw)

Severity: Low Risk

Context: Reward.sol#L187-L188

Description: The finding is not particularly dangerous but it is notable that because `Reward` will allow claiming of rewards on the following Epoch, and because Rebase rewards from the Distributor `Distributor.claim` are distributed based on the balance at the last second of the previous epoch, a desynchronization in how rewards are distributed will happen.

This will end up being fair in the long run however here's an illustrative scenario:

- Locker A has a small lock, they wish to increase the amount they have locked.
- They increase the amount but miss out on rebase rewards (because they are based on their balance at the last second of the previous epoch).
- They decide to `depositManaged` which will distribute rewards based on their current balance, meaning they will "steal" a marginal part of the yield.

- The next epoch, their weight will help increase the yield for everyone, and because Rebasing Rewards are distributed with a week of delay, they will eventually miss out on a similar proportion of yield they "stole".

Recommendation: Consider documenting the behavior in the user documentation.

Velodrome: Managed nfts are intended to be used as long-term vehicles for compounding/farming. We think this is an acceptable trade-off for automation and it does even out over time.

Spearbit: Acknowledged.

6.4.11 `AutoCompounder` can be created without admin

Severity: Low Risk

Context: AutoCompounderFactory.sol#L40

Description: Creating an `AutoCompounder` contract without an `_admin` by passing `address(0)` through `AutoCompounderFactory` is possible. This will break certain functionalities in the AutoCompounder.

Recommendation: It is recommended to add a check to the `_admin` parameter.

Velodrome: Fix commit 74130d.

Spearbit: Fixed.

6.4.12 `claim` and `claimMany` functions will revert when called in end lock time

Severity: Low Risk

Context: RewardsDistributor.sol#L136, RewardsDistributor.sol#L161

Description: If `_timestamp == _locked.end`, then `depositFor()` will be called but this will revert as `block.timestamp >= oldLocked.end`.

Recommendation: It is recommended to change the validation for the following code

```
- if (_timestamp > _locked.end && !_locked.isPermanent) {
+ if (_timestamp - 1 > _locked.end && !_locked.isPermanent) {
```

Velodrome: Fix commit 3aa5bf. Opted to keep the same value comparison of VotingEscrow and used `if(timestamp >= _locked.end ...`

Spearbit: Fixed.

6.4.13 Malicious Pool Factory can be used to prevent new pools from being voted on as well as brick voting locks

Severity: Low Risk

Context: Voter.sol#L322-L381

Description: Because `gauges[_pool]` can only be set once in the voter, governance has the ability to introduce a malicious factory, that will revert on command as a way to prevent normal protocol functionality as well as prevent depositors that voted on these from ever being able to unlock their NFTs

- `ve.withdraw` requires not having voted.
- To remove `voting reset` is called, which in turn calls `IReward(gaugeToFees[gauges[_pool]])._withdraw(uint256(_votes), _tokenId);`.
- If a malicious `gaugeToFees` contract is deployed, the `tokenId` won't be able to ever set `voted` to `false` preventing the ability from ever withdrawing.

Recommendation: It's worth ensuring that no such malicious factory is introduced, which can be achieved by setting up proper gauges initially for most pools.

To ensure this doesn't happen in the future, initially, vetoing governance proposals may be necessary until sufficient decentralization is achieved.

Velodrome: It is worth acknowledging the risks of governance and upgradeability within Voter. There is a possibility of a "bad" gauge being created which could freeze the votes of a veNFT, and new gauge creation requires the diligence of the team and users to prevent this exploit from happening. Aside of the assumptions mentioned above, there are several steps already in place to additionally support the intended experience of the protocol:

- Fallback factories to ensure there is always a trusted gauge for a set of tokens.
- FE management by team to display only legitimate gauges.

Monitoring is definitely needed to ensure governance works as intended. Fortunately, in this case, even after a "bad" gauge is created, a user is not at risk until they submit a transaction depositing into the new gauge.

Spearbit: Acknowledged.

6.4.14 Pool will stop working if a pausable / blockable token is blocked

Severity: Low Risk

Context: Pool.sol

Description: Some tokens are pausable or implement a block list (e.g. USDC), if such a token is part of a Pool, and the Pool is blocked, the Pool will stop working.

It's important to notice that the LP token, which wraps a deposit will still be transferable and the composability with Gauges and Reward Contracts will not be broken even when the pool is unable to function.

Recommendation: It may be best to disclose such risks in the documentation, however, these are inherent risks with using such tokens (USDC included) and are not specifically attributable to the code in-scope.

Velodrome: Acknowledged. Pausing a token would prevent LPs from withdrawing, freeze swaps, and block all claimable rewards of the token.

Spearbit: Acknowledged.

6.5 Gas Optimization

6.5.1 Use ClonesWithImmutableArgs in `AutoCompounderFactory` saves gas

Severity: Gas Optimization

Context: AutoCompounderFactory.sol#L40

Description: The `AutoCompounderFactory` can utilize `ClonesWithImmutableArgs` to deploy new `AutoCompounder` contracts. This would save a lot of gas compared to the current implementation.

Recommendation: It is recommended to use the `ClonesWithImmutableArgs` library

Spearbit: This is an audit that was done by Spearbit using the same library: spearbit-audit-amm.pdf For the sake of transparency a recent exploit related to upgrades and clones was found in Astaria, it's worth checking the post-mortem once made available: AstariaXYZ twitter

Velodrome: Okay, thank you for sharing. We will leave the code as-is. No gas optimization is worth any change in protocol security.

Spearbit: Acknowledged.

6.5.2 Convert hardcoded route to internal function in `CompoundOptimizer`

Severity: Gas Optimization

Context: CompoundOptimizer.sol#L37-L75

Description: All of the hardcoded route setups can be converted to an internal function with hardcoded values.

Recommendation: The recommendation is to convert all of the hardcoded route setups to an internal function with hardcoded values.

Velodrome: Implemented commit 505626.

Spearbit: Fixed.

6.5.3 Early return in `supplyAt` save gas

Severity: Gas Optimization

Context: BalanceLogicLibrary.sol#L120-L122

Description: To save gas, you can return in case of `_epoch` is equal to zero can be made before cache `_point`.

Recommendation: It is recommended to return before cache `_point`.

Velodrome: Code change commit d323b0

Spearbit: Fixed.

6.6 Informational

6.6.1 Approved User could Split NFTs and be unable to continue operating

Severity: Informational

Context: VotingEscrow.sol#L973-L974

Description: An approved user can be approved via `approve`, the storage value set is `idToApprovals[_tokenId] = _approved;`

Splitting will create two new NFTs that will be sent to the `owner`.

This means that an `approved` user would be able to split the NFTs on behalf of the `owner`, however, in doing so they would lose ownership of the NFTs, being unable to continue using them during the TX

Recommendation: This is at most a gotcha to integrators, so there's no particular risk.

Velodrome: Updated docs commit d912a1.

Spearbit: Fixed.

6.6.2 Add `sweep` function to `CompoundOptimizer`

Severity: Informational

Context: CompoundOptimizer.sol

Description: Some tokens may be completely illiquid, may not be worth auto-compounding so it would be best to also allow a way to `sweep` tokens out to the owner for some tokens. Examples:

- Airdrops / Extra rewards.
- Very new tokens that the owner wants to farm instead of dump.

Recommendation: It is recommended to create a `sweep` functionality.

Velodrome: We will implement a sweep solution in which the team manages a list of tokens that cannot be swept. Tokens that cannot be swept are tokens with high liquidity (USDC, DAI, etc.). This list can only be added to, not

removed from, for additional trust. The admin is allowed to sweep any tokens that are not on this list within the first 24 hours after the epoch flip. Trusted keepers will not be able to swap any tokens within the first 24 hours to prevent a race to sweep.

Spearbit: Acknowledged.

6.6.3 Allow Manual Suggestion of Pair in `AutoCompounder`

Severity: Informational

Context: CompoundOptimizer.sol#L79-L82

Description: Allow manual suggestion of token pairs such as USDC, USDT, LUSD, and wBTC. It may be best to pass a list of pairs as parameters to check for additional tokens. Ultimately, if a suggested pair offers a better price, there's no reason not to allow it. The caller should be able to pass a suggested optimal route, which can then be compared against other routes. Use whichever route is best. If the user's suggested route is the best one, use theirs and ensure that the swap goes through.

Recommendation: One possible solution to this problem would be to implement a way for the caller to pass a suggested optimal route, which can then be compared against other routes.

Velodrome: Acknowledged.

Spearbit: Acknowledged.

6.6.4 Check if owner exists in `split` function

Severity: Informational

Context: VotingEscrow.sol#L955

Description: In case the NFT does not exist, the `_ownerOf(_from)` function returns the zero address. This check is satisfied if `canSplit` has been toggled. However, this does not lead to any issues because the `_isApprovedOrOwner()` check will revert as intended, and there is no amount in the lock. It may be a good idea to update the `_ownerOf()` function to revert if there is no owner for the NFT.

Recommendation: It is recommended to check the `ownerOf` different from address(0)

```
 function split(
  uint256 _from,
  uint256 _amount
 ) external nonReentrant returns (uint256 _tokenId1, uint256 _tokenId2) {
  address sender = _msgSender();
  address owner = _ownerOf(_from);
+ if(owner == address(0)) revert SplitNoOwner
  if (!canSplit[owner] && !canSplit[address(0)]) revert SplitNotAllowed();
  ...
 }
```

Velodrome: Implemented commit 197c8d.

Spearbit: Fixed.

6.6.5 Velo and Veto Governor do not use MetaTX Context

Severity: Informational

Context: Velo.sol#LL5C67-L5C68

Description: These two contracts use `Context` instead of `ERC2771Context`.

Recommendation: At this time I'd recommend not to change the code, Velo can still be moved via permit, which is effectively the same as allowing metaTXs.

Velodrome: Acknowledged.

Spearbit: Acknowledged.

6.6.6 SinkManager is depositing to Gauge without using the TokenId

Severity: Informational

Context: SinkManager.sol#L238

Description: `gauge.deposit` allows to specify a tokenId, but the field is unused

Recommendation: I believe a nofix to completely fine.

Velodrome: Acknowledged. There is no risk in leaving the argument empty. The v1 gauge takes in the token argument Gauge.sol#L457 only to populate the `tokenIds` mapping within the Gauge (which is never utilized) and ensure the gauge is alive via `voter.attachTokenToGauge()` (which it will be). Will leave as-is.

Spearbit: Acknowledged.