KittenSwap Security Review

Pashov Audit Group

Conducted by: Said, ast3ros, jesjupyter, t.aksoy

May 7th 2025 - May 22th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Kittenswap/contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About KittenSwap

KittenSwap is a DEX with custom gauge and factory modifications, deployed on HyperEVM with support for LayerZero and hyperlane-wrapped bridged tokens. The audit focused on the voting escrow token locking system, voter-controlled gauge weight distribution, external bribe reward mechanisms, and concentrated liquidity pool factory management.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. 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

6. Security Assessment Summary

review commit hash - b191141c1efa9eda997c75460dab383db878e2b1

fixes review commit hash - bb528db2ab79290f64788cb5a65f1e58cd3aa182

Scope

The following smart contracts were in scope of the audit:

- `Voter`
- `VotingEscrow`
- `Gauge`
- `CLFactory`
- `FactoryRegistry`
- `CLGauge`
- `CLGaugeFactory`
- `ExternalBribe`

7. Executive Summary

Over the course of the security review, Said, ast3ros, jesjupyter, t.aksoy engaged with KittenSwap to review KittenSwap. In this period of time a total of **55** issues were uncovered.

Protocol Summary

Protocol Name	KittenSwap
Repository	https://github.com/Kittenswap/contracts
Date	May 7th 2025 - May 22th 2025
Protocol Type	DEX

Findings Count

Severity	**Amount**
Critical	2
High	7
Medium	16
Low	30
Total Findings	**55**

Summary of Findings

ID	Title	Severity	Status
[C-01]	split can be abused to create locked data with an arbitrary amount	Critical	Resolved
[C-02]	CLGauge sends KITTEN rewards to itself instead of to stakers	Critical	Resolved
[H-01]	Incorrect next epoch supply usage affects reward calculation	High	Resolved
[H-02]	Loss of claimable rewards upon gauge deactivation	High	Resolved
[H-03]	ExternalBribe.earned skips rewards before the last tokenId checkpoint	High	Resolved
[H-04]	Duplicate tokenId in delegate list may inflate votes	High	Resolved
[H-05]	Incorrect last_point.blk calculation due to shared memory in _checkpoint()	High	Resolved
[H-06]	Lack of access control in CLGauge creation allows unauthorized Pools	High	Resolved
[H-07]	Inconsistent rounding of lock time causes voting power errors	High	Resolved
[M-01]	Inconsistent VotingEscrow implementation in balanceOfNFT() methods	Medium	Acknowledged
[M-02]	Ambiguous condition in fee handling for bribe rewards	Medium	Acknowledged
[M-03]	Weight loss possible due to invalid pool votes in _vote()	Medium	Resolved

[M-04]	checkpoint.timestamp inside VotingEscrow never used	Medium	Resolved
[M-05]	Missing rescue reward function in Gauge, CLGauge, Bribes	Medium	Resolved
[M-06]	VotingEscrow._burn prevents approved spender from using withdraw()	Medium	Resolved
[M-07]	VotingEscrow.withdraw cannot be called by an approved tokenId spender	Medium	Resolved
[M-08]	Wrong supply accounting in merge function	Medium	Resolved
[M-09]	Desync lets bribes be received without emissions	Medium	Resolved
[M-10]	split() mints NFTs to msg.sender not the original owner	Medium	Resolved
[M-11]	User can prevent poke() from updating voting power	Medium	Resolved
[M-12]	Merging or withdrawing VotingEscrow burns tokens without rewards	Medium	Resolved
[M-13]	Potential revert in earned() when checkpoint supply is zero	Medium	Resolved
[M-14]	Incorrect check inside CLGauge.claimFees	Medium	Resolved
[M-15]	DOS attack by delegating tokens at MAX_DELEGATES = 1024	Medium	Resolved
[M-16]	Double reward claim in ExternalBribe	Medium	Acknowledged

[L-01]	Redundant logic in getEpochStart() of ExternalBribe	Low	Acknowledged
[L-02]	No revert in ownerOf() for non-existent token IDs	Low	Resolved
[L-03]	Inadequate reward validation in ClGauge contract	Low	Acknowledged
[L-04]	CLGauge should use ERC721HolderUpgradeable instead of ERC721Holder	Low	Resolved
[L-05]	Irreversible whitelisting of tokens	Low	Resolved
[L-06]	Redundant check for _votes in _reset() function	Low	Resolved
[L-07]	Hardcoded gas in ClFactory's getSwapFee and getUnstakedFee can cause issues	Low	Acknowledged
[L-08]	Voter.updateForRange could revert if end value is invalid	Low	Acknowledged
[L-09]	Potential liquidity overflow	Low	Resolved
[L-10]	Fee-on-transfer token incompatibility	Low	Acknowledged
[L-11]	Reward calculation reverts due to potential arithmetic overflow	Low	Resolved
[L-12]	Initializer pattern in ClFactory lacks upgrade, risks frontrunning	Low	Resolved
[L-13]	Users can create NFTs with 0 locked amount through split function	Low	Resolved
[L-14]	Gauge may receive rewards for period it was dead when revived	Low	Resolved
[L-15]	Rewards lost until Gauge or Bribe deposits are nonzero	Low	Acknowledged

[L-16]	Dust losses in notifyRewardAmount	Low	Acknowledged
[L-17]	Lack of _disableInitializers in upgradeable contracts	Low	Resolved
[L-18]	Potential out of gas risk in reward distribution checkpoint traversal	Low	Acknowledged
[L-19]	Inaccuracy in Voter contract reward distribution mechanism	Low	Acknowledged
[L-20]	Calling notifyRewardAmount for unsupported tokens can grief gauge	Low	Resolved
[L-21]	Incorrect logic and documentation in balance retrieval	Low	Acknowledged
[L-22]	Voting restriction possible after resetting in reset()	Low	Acknowledged
[L-23]	VotingEscrow.totalSupplyAtT is not working properly	Low	Acknowledged
[L-24]	Inconsistent restriction on increase_amount and deposit_for	Low	Acknowledged
[L-25]	Lack of a _poolVote length check inside Voter.vote could cause issues	Low	Acknowledged
[L-26]	Users lose rewards if they call getReward exactly at epoch end	Low	Acknowledged
[L-27]	Users can deposit NFTs into killed CL gauges	Low	Resolved
[L-28]	Inaccurate reward calculation in ExternalBribe contract	Low	Acknowledged
[L-29]	CLGauge.earned could revert due to reward growth underflow	Low	Resolved
[L-30]	Inconsistent DOMAIN_TYPEHASH	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] `split` can be abused to create `locked` data with an arbitrary `amount`

Severity

Impact: High

Likelihood: High

Description

Due to the lack of a check on the split amount inside `split`, a user can provide an arbitrary `_amount` to the function. Since `locked.amount` is stored as an `int128`, this can result in one of the split tokens having a negative value, while the other is set to the full `_amount` provided.

```
function split(
        uint _from,
        uint _amount
    ) external returns (uint _tokenId1, uint _tokenId2) {
        address msgSender = msg.sender;

        require(_isApprovedOrOwner(msgSender, _from));
        require(attachments[_from] == 0 && !voted[_from], "attached");
        require(_amount > 0, "Zero Split");

        // burn old NFT
        LockedBalance memory _locked = locked[_from];
        int128 value = _locked.amount;
        locked[_from] = LockedBalance(0, 0);
        _checkpoint(_from, _locked, LockedBalance(0, 0));
        _burn(_from);

        // set max lock on new NFTs
        _locked.end = block.timestamp + MAXTIME;

        int128 _splitAmount = int128(uint128(_amount));
        // @audit - underflow is possible
>>>     _locked.amount = value - _splitAmount; // already checks for underflow
// here in ^0.8.0
        _tokenId1 = _createSplitNFT(msgSender, _locked);

>>>     _locked.amount = _splitAmount;
        _tokenId2 = _createSplitNFT(msgSender, _locked);

        emit Split(
            _from,
            _tokenId1,
            _tokenId2,
            msgSender,
            uint(uint128(locked[_tokenId1].amount)),
            uint(uint128(_splitAmount)),
            _locked.end,
            block.timestamp
        );
    }
```

This can be abused by initially depositing a small amount of tokens, then calling `split` with an arbitrary amount to create a large `locked` data. Then can be withdrawn at the end of the lock period, and also grants an arbitrary amount of voting power and balance, which can be used to claim rewards from gauges and bribes.

PoC :

```
function testSplitVeKittenUnderflow() public {
        testDistributeVeKitten();

        vm.startPrank(user1);

        uint veKittenId = veKitten.tokenOfOwnerByIndex(user1, 0);
        (int128 lockedAmount, uint endTime) = veKitten.locked(veKittenId);

        (uint t1, uint t2) = veKitten.split(veKittenId, uint256(uint128
          (lockedAmount)) * 100);

        (int128 lockedAmountSplit, ) = veKitten.locked(t2);

        console.log("initial token to split locked amount :");
        console.log(lockedAmount);
        console.log("splitted token locked amount :");
        console.log(lockedAmountSplit);

        vm.stopPrank();
    }
```

Log output :

```
Logs:
  initial token to split locked amount :
  100000000000000000000000000
  splitted token locked amount :
  100000000000000000000000000
```

Recommendations

Validate that the provided `_amount` does not exceed the locked amount of the split token.

[C-02] CLGauge sends KITTEN rewards to itself instead of to stakers

Severity

Impact: High

Likelihood: High

Description

When users stake their NFP tokens, ownership of these NFTs is transferred to the CLGauge contract itself. When a staker claims their KITTEN rewards, the `_getReward` function incorrectly sends these rewards to the current owner of

the NFP (which is the CLGauge contract) rather than to the original staker who is entitled to these rewards.

In the `_getReward` function below, we can see that the rewards are being sent to the NFP owner, which after staking is the CLGauge contract itself:

```
function _getReward(uint256 nfpTokenId) internal {
        (, , , , , int24 _tickLower, int24 _tickUpper, , , , , ) = nfp
            .positions(nfpTokenId);

        _updateRewardForNfp(nfpTokenId, _tickLower, _tickUpper);

        uint256 reward = rewards[nfpTokenId];
        address owner = nfp.ownerOf
        //(nfpTokenId); // @audit owner is CLGauge contract

        if (reward > 0) {
            delete rewards[nfpTokenId];
            _safeApprove(kitten, address(this), reward);
            _safeTransferFrom(kitten, address(this), owner, reward);
            emit ClaimRewards(owner, reward);
        }
    }
```

It leads to 100% of staked users' rewards are lost and the rewards are not recoverable once sent to the gauge contract.

Recommendations

Modify the `_getReward` function to accept the actual staker's address as a parameter rather than deriving it from NFP ownership:

```
-    function _getReward(uint256 nfpTokenId) internal {
+    function _getReward(uint256 nfpTokenId, address owner) internal {
        ...
-       address owner = nfp.ownerOf(nfpTokenId);

        if (reward > 0) {
            delete rewards[nfpTokenId];
-            _safeApprove(kitten, address(this), reward);
-            _safeTransferFrom(kitten, address(this), owner, reward);
-            emit ClaimRewards(owner, reward);
+            _safeTransfer(kitten, owner, reward);
+            emit ClaimRewards(owner, reward);
        }
    }
```

8.2. High Findings

[H-01] Incorrect next epoch supply usage affects reward calculation

Severity

Impact: High

Likelihood: Medium

Description

Context: This is an issue of the forked repo that is not being fixed. (https://github.com/spearbit/portfolio/blob/master/pdfs/Velodrome-Spearbit-Security-Review.pdf 5.1.1 Reward calculates earned incorrectly on each epoch boundary).

In the `ExternalBribe` contract, the following line is used to retrieve the previous epoch's supply:

```
_prev._prevSupply = supplyCheckpoints[getPriorSupplyIndex
  (_nextEpochStart + DURATION)].supply;
```

Here, `_nextEpochStart + DURATION` is used to determine the index for retrieving the supply. This means that the contract is actually accessing the supply from the next epoch rather than the previous one.

Recommendations

Use `_nextEpochStart + DURATION-1`.

[H-02] Loss of claimable rewards upon gauge deactivation

Severity

Impact: High

Likelihood: Medium

Description

The `killGauge` and `reviveGauge` functions in the `Voter` contract allow the `emergencyCouncil` to deactivate and reactivate gauges. However, once a gauge is killed, its associated claimable rewards are set to zero, resulting in the permanent loss of any accumulated rewards, even if the gauge is later revived.

In the contract, the following code snippets illustrate the issue:

```
function killGauge(address _gauge) external {
       require(msg.sender == emergencyCouncil, "not emergency council");
       require(isAlive[_gauge], "gauge already dead");
       isAlive[_gauge] = false;
       claimable[_gauge] = 0; // Claimable rewards are reset to zero
       emit GaugeKilled(_gauge);
   }
```

When a gauge is killed, the `claimable` rewards for that gauge are set to zero, meaning that any rewards that were accumulated prior to killing the gauge are permanently lost. Even if the gauge is revived later, the previously accumulated rewards cannot be recovered.

Also, at the same time, the other gauge's reward is still based on `totalWeight`, which is never properly adjusted, leading to a loss of rewards.

```
uint _delta = _index - _supplyIndex; // see if there is any
            // difference that need to be accrued
            if (_delta > 0) {
                uint _share = (uint
                //(_supplied) * _delta) / 1e18; // add accrued difference for each sup
                if (isAlive[_gauge]) {
                    claimable[_gauge] += _share;
                }
```

```
function notifyRewardAmount(uint amount) external {
        _safeTransferFrom(base, msg.sender, address
        //(this), amount); // transfer the distro in
        uint256 _ratio =
        //(amount * 1e18) / totalWeight; // 1e18 adjustment is removed during claim
        if (_ratio > 0) {
            index += _ratio;
        }
        emit NotifyReward(msg.sender, base, amount);
    }
```

Recommendations

To address this issue, it is recommended to implement a mechanism that allows for the recovery of `claimable` rewards when a gauge is revived.

[H-03] `ExternalBribe.earned` skips rewards before the last `tokenId` checkpoint

Severity

Impact: High

Likelihood: Medium

Description

Inside `ExternalBribe.earned`, it first loops up to `_endIndex - 1` to determine whether to include previous rewards in the earned amount.

```
function earned(address token, uint tokenId) public view returns (uint) {
        uint _startTimestamp = lastEarn[token][tokenId];
        if (numCheckpoints[tokenId] == 0) {
            return 0;
        }

        uint _startIndex = getPriorBalanceIndex(tokenId, _startTimestamp);
        uint _endIndex = numCheckpoints[tokenId] - 1;

        uint reward = 0;
        // you only earn once per epoch (after it's over)
        RewardCheckpoint memory prevRewards;

        prevRewards.timestamp = _bribeStart(_startTimestamp);

        _prev._prevSupply = 1;

        console.log("END INDEX : ");
        console.log(_endIndex);

        if (_endIndex > 0) {
>>>         for (uint i = _startIndex; i <= _endIndex - 1; i++) {
                _prev._prevTs = checkpoints[tokenId][i].timestamp;
                _prev._prevBal = checkpoints[tokenId][i].balanceOf;
                uint _nextEpochStart = _bribeStart(_prev._prevTs);
                // check that you've earned it
                // this won't happen until a week has passed
>>>             if (_nextEpochStart > prevRewards.timestamp) {
                    console.log("REWARD ADDED : ");
                    console.log(prevRewards.balance);
                    reward += prevRewards.balance;
                }

                prevRewards.timestamp = _nextEpochStart;
                _prev._prevSupply = supplyCheckpoints[
                    getPriorSupplyIndex(_nextEpochStart + DURATION)
                ].supply;
                prevRewards.balance =
                    (_prev._prevBal *
                        tokenRewardsPerEpoch[token][_nextEpochStart]) /
                    _prev._prevSupply;
            }
        }

        Checkpoint memory _cp0 = checkpoints[tokenId][_endIndex];
        (_prev._prevTs, _prev._prevBal) = (_cp0.timestamp, _cp0.balanceOf);

        uint _lastEpochStart = _bribeStart(_prev._prevTs);
        uint _lastEpochEnd = _lastEpochStart + DURATION;

        if (
            block.timestamp > _lastEpochEnd && _startTimestamp < _lastEpochEnd
        ) {
            SupplyCheckpoint memory _scp0 = supplyCheckpoints[
                getPriorSupplyIndex(_lastEpochEnd)
            ];
            _prev._prevSupply = _scp0.supply;
            reward += (_prev._prevBal *
                    tokenRewardsPerEpoch[token][_lastEpochStart]) /
                _prev._prevSupply;
        }

        return reward;
    }
```

However, the logic should include `_endIndex`, as the checkpoint before it must be checked, otherwise, it will always be skipped and not counted as a reward.

Add the following test file to the repo: https://gist.github.com/said017/b96daba163bf2e1eb101589bc541ce06.

Run the test :

```
forge test --match-test testEpochCalculationIssue -vvv
```

Log output :

```
Logs:

  Earned at end of first epoch: 10000000000000000000

  Earned at end of second epoch: 0

  Earned at end of third epoch: 10000000000000000000
```

Recommendations

Change the calculation logic to include `_endIndex`.

```
function earned(address token, uint tokenId) public view returns (uint) {
        uint _startTimestamp = lastEarn[token][tokenId];
        if (numCheckpoints[tokenId] == 0) {
            return 0;
        }

        uint _startIndex = getPriorBalanceIndex(tokenId, _startTimestamp);
        uint _endIndex = numCheckpoints[tokenId] - 1;

        uint reward = 0;
        // you only earn once per epoch (after it's over)
        RewardCheckpoint memory prevRewards;

        prevRewards.timestamp = _bribeStart(_startTimestamp);

        _prev._prevSupply = 1;

        console.log("END INDEX : ");
        console.log(_endIndex);

        if (_endIndex > 0) {
-           for (uint i = _startIndex; i <= _endIndex - 1; i++) {
+           for (uint i = _startIndex; i <= _endIndex ; i++) {
               _prev._prevTs = checkpoints[tokenId][i].timestamp;
               _prev._prevBal = checkpoints[tokenId][i].balanceOf;
               uint _nextEpochStart = _bribeStart(_prev._prevTs);
               // check that you've earned it
               // this won't happen until a week has passed
               if (_nextEpochStart > prevRewards.timestamp) {
                   console.log("REWARD ADDED : ");
                   console.log(prevRewards.balance);
                   reward += prevRewards.balance;
               }
/ ...
}
```

[H-04] Duplicate `tokenId` in delegate list may inflate votes

Severity

Impact: High

Likelihood: Medium

Description

The following issue occurs when moving delegates and was already identified in the Spearbit audit of Velodrome.

Inside `_moveAllDelegates` and `_moveTokenDelegates`, the delegated `tokenIds` will be removed from `srcRep` (previous delegatee) and will be added to the

`dstRep` (new delegatee).

```
function _moveAllDelegates(
        address owner,
        address srcRep,
        address dstRep
    ) internal {
        // You can only redelegate what you own
        if (srcRep != dstRep) {
            if (srcRep != address(0)) {
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint[] storage srcRepOld = srcRepNum > 0
                    ? checkpoints[srcRep][srcRepNum - 1].tokenIds
                    : checkpoints[srcRep][0].tokenIds;
                uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
                uint[] storage srcRepNew = checkpoints[srcRep][nextSrcRepNum]
                    .tokenIds;
                // All the same except what owner owns
                for (uint i = 0; i < srcRepOld.length; i++) {
                    uint tId = srcRepOld[i];
>>>                 if (idToOwner[tId] != owner) {
                        srcRepNew.push(tId);
                    }
                }

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

            if (dstRep != address(0)) {
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint[] storage dstRepOld = dstRepNum > 0
                    ? checkpoints[dstRep][dstRepNum - 1].tokenIds
                    : checkpoints[dstRep][0].tokenIds;
                uint32 nextDstRepNum = _findWhatCheckpointToWrite(dstRep);
                uint[] storage dstRepNew = checkpoints[dstRep][nextDstRepNum]
                    .tokenIds;
                uint ownerTokenCount = ownerToNFTokenCount[owner];
                require(
                    dstRepOld.length + ownerTokenCount <= MAX_DELEGATES,
                    "dstRep would have too many tokenIds"
                );
                // All the same
                for (uint i = 0; i < dstRepOld.length; i++) {
                    uint tId = dstRepOld[i];
>>>                 dstRepNew.push(tId);
                }
                // Plus all that's owned
                for (uint i = 0; i < ownerTokenCount; i++) {
                    uint tId = ownerToNFTokenIdList[owner][i];
>>>                 dstRepNew.push(tId);
                }

>>>             numCheckpoints[dstRep] = dstRepNum + 1;
            }
        }
    }
```

`_findWhatCheckpointToWrite` will be used to determine which checkpoint the list will be added / removed.

```
function _findWhatCheckpointToWrite(
        address account
    ) internal view returns (uint32) {
        uint _timestamp = block.timestamp;
        uint32 _nCheckPoints = numCheckpoints[account];

        if (
            _nCheckPoints > 0 &&
            checkpoints[account][_nCheckPoints - 1].timestamp == _timestamp
        ) {
            return _nCheckPoints - 1;
        } else {
            return _nCheckPoints;
        }
    }
```

Notice that if the operation is called within the same block, the same checkpoint will used for the move delegate operation.

This could cause an issue, consider this scenario (all operation is done on the same block).

Alice Num checkpoint = 1 TokenIds (checkpoint 0) = 1 , 2

Bob Num checkpoint = 1 TokenIds (checkpoint 0) = 3, 4

TokenId 1 is moved from alice to bob, resulting in the following state.

Alice Num checkpoint = 2 TokenIds (checkpoint 0) = 1 , 2 TokenIds (checkpoint 1) = 2

Bob Num checkpoint = 2 TokenIds (checkpoint 0) = 3, 4 TokenIds (checkpoint 1) = 3, 4, 1

TokenId 2 moved from alice to bob, but since it performed in the same block, will result in the following state.

Alice Num checkpoint = 3 TokenIds (checkpoint 0) = 1 , 2 TokenIds (checkpoint 1) = 2

Bob Num checkpoint = 3 TokenIds (checkpoint 0) = 3, 4 TokenIds (checkpoint 1) = 3, 4, 1, 3, 4, 1, 2

There are several issues here: the number of checkpoints keeps increasing even when using the same checkpoint, and the tokenId 2 is not removed from Alice's latest checkpoint. And Bob's latest checkpoint contains several duplicates, as it always assumes `dstRepNew` is empty.

Recommendations

Consider adjusting `_moveAllDelegates` and `_moveTokenDelegates` to account for scenarios where the operation is performed within the same block.

[H-05] Incorrect `last_point.blk` calculation due to shared memory in `_checkpoint()`

Severity

Impact: Medium

Likelihood: High

Description

In the `_checkpoint` function of the VotingEscrow contract, there's an issue with how `initial_last_point` is assigned as a reference to `last_point` instead of creating an independent copy. This causes incorrect block number extrapolation calculations which affect voting power calculations throughout the system.

```
function _checkpoint(
        uint _tokenId,
        LockedBalance memory old_locked,
        LockedBalance memory new_locked
    ) internal {
        ...
        Point memory initial_last_point = last_point; // @audit
        // initial_last_point is a pointer which points to last_point instead of a cop
        uint block_slope = 0; // dblock/dt
        if (block.timestamp > last_point.ts) {
            block_slope =
                (MULTIPLIER * (block.number - last_point.blk)) /
                (block.timestamp - last_point.ts);
        }

        {
            uint t_i = (last_checkpoint / WEEK) * WEEK;
            for (uint i = 0; i < 255; ++i) {
                // Hopefully it won't happen that this won't get used in 5
                // years!
                // If it does, users will be able to withdraw but vote weight
                // will be broken
                t_i += WEEK;
                int128 d_slope = 0;
                if (t_i > block.timestamp) {
                    t_i = block.timestamp;
                } else {
                    d_slope = slope_changes[t_i];
                }
                last_point.bias -=
                    last_point.slope *
                    int128(int256(t_i - last_checkpoint));
                last_point.slope += d_slope;
                if (last_point.bias < 0) {
                    // This can happen
                    last_point.bias = 0;
                }
                if (last_point.slope < 0) {
                    // This cannot happen - just in case
                    last_point.slope = 0;
                }
                last_checkpoint = t_i;
                last_point.ts = t_i; // @audit last_point.ts is updated to t_i
                last_point.blk =
                    initial_last_point.blk +
                    (block_slope *
                    //(t_i - initial_last_point.ts)) / // @audit the initial_last_poin
                    MULTIPLIER;
                _epoch += 1;
                if (t_i == block.timestamp) {
                    last_point.blk = block.number;
                    break;
                } else {
                    point_history[_epoch] = last_point;
                }
            }
        }

        ...
    }
```

When `last_point.ts` is updated to `t_i` in the loop, `initial_last_point.ts` is also updated to the same value because they reference the same memory

location. This results in `(t_i - initial_last_point.ts)` evaluating to zero, causing `last_point.blk` to remain unchanged throughout the loop iterations.

This issue directly impacts:

- The accuracy of historical voting power calculations: Total voting power and Per-NFT voting power.
- Governance functionality that depends on accurate vote weight.

Recommendations

Create a copy of the `last_point`:

```
function _checkpoint(
        uint _tokenId,
        LockedBalance memory old_locked,
        LockedBalance memory new_locked
    ) internal {
        ...
-       Point memory initial_last_point = last_point;
+       Point memory initial_last_point = Point({
+         bias: last_point.bias,
+         slope: last_point.slope,
+         ts: last_point.ts,
+         blk: last_point.blk
        ...
});
```

[H-06] Lack of access control in `CLGauge` creation allows unauthorized `Pools`

Severity

Impact: Medium

Likelihood: High

Description

The `CLGaugeFactory.createGauge` function can be called by anyone, which creates a front-running vulnerability. An attacker can monitor the mempool for legitimate `Voter.createCLGauge` transactions and front-run them, calling `CLGaugeFactory.createGauge` with the same pool address before the transaction from Voter contract is processed.

```
function createGauge(
        address _pool,
        address _internal_bribe,
        address _kitten,
        bool _isPool
    ) external returns (address) {
        CLGauge newGauge = CLGauge(
            address(
                new ERC1967Proxy(
                    implementation,
                    abi.encodeCall(
                        CLGauge.initialize,
                        (
                            _pool,
                            _internal_bribe,
                            _kitten,
                            ve,
                            voter,
                            nfp,
                            _isPool
                        )
                    )
                )
            )
        );

        ICLPool(_pool).setGaugeAndPositionManager(address(newGauge), nfp);

        ...
    }
```

This function calls `ICLPool(_pool).setGaugeAndPositionManager(address(newGauge), nfp)`, which can only be called once per pool due to the following check in the CLPool.setGaugeAndPositionManager function:

```
function setGaugeAndPositionManager(
        address _gauge,
        address _nft
    ) external override lock onlyGaugeFactory {
        require(gauge == address(0)); // @audit Can only be called once
        gauge = _gauge;
        nft = _nft;
    }
```

The problem arises when the legitimate `Voter.createCLGauge` function is called later, which attempts to create a gauge through the same factory. The function will revert.

```
function createCLGauge(
        address _poolFactory,
        address _pool
    ) external returns (address) {
        // ...
        address _gauge = ICLGaugeFactory(gaugefactory).createGauge(
            _pool,
            _internal_bribe,
            base,
            isPool
        );
        // ...
    }
```

This effectively creates a denial of service for the intended gauge creation process, disrupting the protocol's governance and reward distribution mechanisms.

Recommendations

Add access control to restrict the `createGauge` function to only be callable by the voter contract:

```
function createGauge(
        address _pool,
        address _internal_bribe,
        address _kitten,
        bool _isPool
    ) external returns (address) {
+       require(msg.sender == voter, "Only voter can create gauge");
        CLGauge newGauge = CLGauge(
            address(
                new ERC1967Proxy(
                    implementation,
                    abi.encodeCall(
                        CLGauge.initialize,
                        (
                            _pool,
                            _internal_bribe,
                            _kitten,
                            ve,
                            voter,
                            nfp,
                            _isPool
                        )
                    )
                )
            )
        );

        ...
    }
```

[H-07] Inconsistent rounding of lock time causes voting power errors

Severity

Impact: Medium

Likelihood: High

Description

In the VotingEscrow contract, lock end times are consistently rounded down to week boundaries (Thursday 00:00 UTC) in all functions except for `split`. This inconsistency disrupts the checkpoint system and causes incorrect voting power calculations.

In `_create_lock` and `increase_unlock_time`, lock end time is explicitly rounded to weekly boundaries. However, in the `split` function, this rounding is missing:

```
function split(
        uint _from,
        uint _amount
    ) external returns (uint _tokenId1, uint _tokenId2) {
        ...
        // set max lock on new NFTs
        _locked.end = block.timestamp + MAXTIME; // @audit the _locked.end isn't
        // rounded down to week
        ...
    }
```

This inconsistency has severe implications because the contract's checkpoint mechanism relies on week aligned timestamps when applying slope changes:

```
uint t_i = (last_checkpoint / WEEK) * WEEK;
    for (uint i = 0; i < 255; ++i) {
        t_i += WEEK;
        int128 d_slope = 0;
        if (t_i > block.timestamp) {
            t_i = block.timestamp;
        } else {
            d_slope = slope_changes[t_i]; // @audit t_i is aways on Thursday
        }
        // ...voting power calculations continue...
}
```

When a lock expires, its slope should be removed from the total slope to decrease voting power. However, if the lock expiration time is not aligned with a weekly boundary, the checkpoint system will never process the slope change at the exact expiration time. The global slop will always be wrong.

Users or attackers can call `split` to break the checkpoint calculation of the contract. This results in inflated voting power and incorrect reward distributions based on overstated voting weights.

Recommendations

Modify the split function to align lock end times with weekly boundaries, consistent with the rest of the codebase:

```
function split(
        uint _from,
        uint _amount
    ) external returns (uint _tokenId1, uint _tokenId2) {
        ...
        // set max lock on new NFTs
-        _locked.end = block.timestamp + MAXTIME;
+        _locked.end = ((block.timestamp + MAXTIME) / WEEK) * WEEK;
        ...
    }
```

8.3. Medium Findings

[M-01] Inconsistent `VotingEscrow` implementation in `balanceOfNFT()` methods

Severity

Impact: Medium

Likelihood: Medium

Description

This is an unfixed issue in the forked repo (link, 5.3.9 Inconsistent between balanceOfNFT, balanceOfNFTAt and _balanceOfNFT functions).

> 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.

Recommendations

Keep the consistency. It seems like the check is no longer in use, if so, remove it from the codebase.

[M-02] Ambiguous condition in fee handling for bribe rewards

Severity

Impact: Medium

Likelihood: Medium

Description

The condition requiring `_fees0`/`_fee1` to be greater than the amount returned by `IBribe(internal_bribe).left(_token0)`/`IBribe(internal_bribe).left(_token1)` in the `Gauge` contract is unclear and potentially misleading. This condition does not establish a necessary relationship between the fees being processed and the remaining rewards available for distribution.

In the `Gauge` contract, the following code snippet checks the relationship between `_fees0` and the amount left in the bribe contract:

```
if (
    _fees0 > IBribe(internal_bribe).left(_token0) &&
    _fees0 / DURATION > 0
) {
    fees0 = 0;
    _safeApprove(_token0, internal_bribe, _fees0);
    IBribe(internal_bribe).notifyRewardAmount(_token0, _fees0);
}
```

The condition _fees0 > IBribe(internal_bribe).left(_token0) is problematic because:

- Ambiguity: The `_fees0` variable represents new rewards that are being processed and will be passed to `notifyRewardAmount`, while `left` indicates the amount of rewards that have not yet been distributed. There is no inherent connection between these two values, making the condition unclear.
- Logical Disconnect: The requirement implies that the new fees must exceed the remaining rewards, which does not logically correlate with the process of notifying the bribe contract about new rewards.

Note: Also, the check `_claimable/DURAITON` is a bit too strict compared with what was checked in the `CLGauge` which uses `epochDurationLeft` instead of `DURATION`:

```
if (
            _claimable > IGauge(_gauge).left(base) && _claimable / DURATION > 0
        ) {
            claimable[_gauge] = 0;
            IGauge(_gauge).notifyRewardAmount(base, _claimable);
            emit DistributeReward(msg.sender, _gauge, _claimable);
        }
```

```
if (block.timestamp >= periodFinish) {
            rewardRate = amount / epochDurationLeft;
            pool.syncReward({
                rewardRate: rewardRate,
                rewardReserve: amount,
                periodFinish: nextPeriodFinish
            });
        } else {
            uint256 newAmount = amount + epochDurationLeft * rewardRate;
            rewardRate = newAmount / epochDurationLeft;
            pool.syncReward({
                rewardRate: rewardRate,
                rewardReserve: newAmount,
                periodFinish: nextPeriodFinish
            });
        }
```

Recommendations

To improve clarity and ensure correct functionality, it is recommended to revise the condition to better reflect the intended logic.

[M-03] Weight loss possible due to invalid pool votes in `_vote()`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `_vote` function of the `Voter` contract, if the provided `_poolVote` array contains invalid pools, the calculation of `_totalVoteWeight` still uses the sum of all `weights`. This can lead to inefficient weight allocation and potential loss of rewards for valid gauges.

Within the `_vote` function, the following code snippet is executed:

```
uint256 _weight = IVotingEscrow(_ve).balanceOfNFT(_tokenId);
uint256 _totalVoteWeight = 0;

for (uint i = 0; i < _poolCnt; i++) {
    _totalVoteWeight += _weights[i];
}

for (uint i = 0; i < _poolCnt; i++) {
    address _pool = _poolVote[i];
    address _gauge = gauges[_pool];

    if (isGauge[_gauge]) {
        uint256 _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;
        // ... additional logic ...
    }
}
```

If the `_poolVote` contains pools that are not valid gauges(not added before), the function skips the calculation for those pools without reverting. As a result, **the valid gauges that are processed will use the potentially inflated `_totalVoteWeight` as a divisor, leading to a miscalculation of `_poolWeight`.** This means that valid votes may not utilize the actual weight of the NFT, resulting in a loss of potential rewards.

This issue can lead to significant inefficiencies in the voting process. If users input incorrect pool votes, a substantial portion of their voting weight may be wasted, leading to reduced rewards. Since the `vote` function is protected by the `onlyNewEpoch` modifier, users cannot correct their mistakes within the same epoch, compounding the potential loss.

Consider the following scenario:

User Input: The user provides the following inputs:

- `_poolVote`: An array of pools the user wants to vote on: `["PoolA", "PoolB", "PoolC"]`.
- `_weights`: The corresponding weights for each pool: `[50, 50, 50]`.
- Gauge Status: `gauges["PoolA"]` is a valid gauge, `gauges["PoolB"]` is a valid gauge, `gauges["PoolC"]` is not a valid gauge.
- NFT Weight: The user's NFT weight is retrieved as: `_weight = IVotingEscrow(_ve).balanceOfNFT(_tokenId);` (let's say this returns `150`).

As a result:

- Total Weight Used: The user intended to vote with NFT weight which is 150.
- Actual Weight Utilized: Only 100 (from `PoolA` and `PoolB`) is effectively utilized:
- Weight Waste: The weight associated with `PoolC` (the invalid gauge) is wasted.

Recommendations

To mitigate this issue, it is recommended to implement a validation check for the `_poolVote` array before proceeding with the weight calculations. If any pools are found to be invalid, the function should revert or adjust the calculations accordingly.

[M-04] `checkpoint.timestamp` inside `VotingEscrow` never used

Severity

Impact: Medium

Likelihood: Medium

Description

The design of `checkpoint` in `VotingEscrow` is to record the delegatee's information at the time of execution. This is necessary for optimization and for querying how much a `tokenId` was delegated to a specific address at a given time.

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

        // Next check implicit zero balance
        if (checkpoints[account][0].timestamp > timestamp) {
            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.timestamp == timestamp) {
                return center;
            } else if (cp.timestamp < timestamp) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return lower;
    }

    function getPastVotes(
        address account,
        uint timestamp
    ) public view returns (uint) {
        uint32 _checkIndex = getPastVotesIndex(account, timestamp);
        // Sum votes
        uint[] storage _tokenIds = checkpoints[account][_checkIndex].tokenIds;
        uint votes = 0;
        for (uint i = 0; i < _tokenIds.length; i++) {
            uint tId = _tokenIds[i];
            // Use the provided input timestamp here to get the right decay
            votes = votes + _balanceOfNFT(tId, timestamp);
        }
        return votes;
    }
```

However, `checkpoints.timestamp` is never used and initialized, causing all operations that depend on this information will not work properly.

Recommendations

When delegate information is moved, initialize and set `checkpoints.timestamp`.

[M-05] Missing rescue reward function in `Gauge`, `CLGauge`, `Bribes`

Severity

Impact: High

Likelihood: Low

Description

When rewards are distributed to `Gauge`, `CLGauge`, and Bribes contracts via `notifyRewardAmount`, it's possible that the total supply is 0 during that reward period/epoch, resulting in no rewards being distributed.

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

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

        periodFinish[token] = adjustedTstamp + DURATION;

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

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

However, those contracts lack functionality to rescue the rewards, causing the distributed rewards to become permanently stuck inside them.

Recommendations

Add functionality to rescue the rewards.

[M-06] `VotingEscrow._burn` prevents approved spender from using `withdraw()`

Severity

Impact: Medium

Likelihood: Medium

Description

Inside the `withdraw`, `merge`, and `split` operations, there is a check using `_isApprovedOrOwner`, which allows the `tokenId`'s spender or operator to perform those actions.

```
function withdraw(uint _tokenId) external nonreentrant {
>>>     assert(_isApprovedOrOwner(msg.sender, _tokenId));
        require(attachments[_tokenId] == 0 && !voted[_tokenId], "attached");

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

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

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

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

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

        emit Withdraw(msg.sender, _tokenId, value, block.timestamp);
        emit Supply(supply_before, supply_before - value);
    }
```

However, inside the `_burn` function, it attempts to remove the token from `msg.sender` instead of the actual `owner`.

```
function _burn(uint _tokenId) internal {
        require(
            _isApprovedOrOwner(msg.sender, _tokenId),
            "caller is not owner nor approved"
        );

        address owner = ownerOf(_tokenId);

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

```
function _removeTokenFrom(address _from, uint _tokenId) internal {
        // Throws if `_from` is not the current owner
>>>     assert(idToOwner[_tokenId] == _from);
        // Change the owner
        idToOwner[_tokenId] = address(0);
        // Update owner token index tracking
        _removeTokenFromOwnerList(_from, _tokenId);
        // Change count tracking
        ownerToNFTokenCount[_from] -= 1;
    }
```

This will prevent `tokenId`'s spender and operator to perform those actions on behalf of owner.

Recommendations

Provide `owner` instead of `msg.sender` to `_removeTokenFrom` operation inside `_burn`.

[M-07] `VotingEscrow.withdraw` cannot be called by an approved `tokenId` spender

Severity

Impact: Medium

Likelihood: Medium

Description

Inside `VotingEscrow.withdraw`, there is `_isApprovedOrOwner` check, which allows an approved `tokenId` spender to trigger this operation.

```
function withdraw(uint _tokenId) external nonreentrant {
>>>     assert(_isApprovedOrOwner(msg.sender, _tokenId));
        require(attachments[_tokenId] == 0 && !voted[_tokenId], "attached");

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

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

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

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

        // Burn the NFT
        _burn(_tokenId);

        emit Withdraw(msg.sender, _tokenId, value, block.timestamp);
        emit Supply(supply_before, supply_before - value);
    }
```

```
function _isApprovedOrOwner(
        address _spender,
        uint _tokenId
    ) internal view returns (bool) {
        address owner = idToOwner[_tokenId];
        bool spenderIsOwner = owner == _spender;
>>>     bool spenderIsApproved = _spender == idToApprovals[_tokenId];
        bool spenderIsApprovedForAll = (ownerToOperators[owner])[_spender];
        return spenderIsOwner || spenderIsApproved || spenderIsApprovedForAll;
    }
```

However, inside `_burn`, when `approve` is called to reset `tokenId` approval, there is a check that prevent the approved spender to perform the operation.

```
function approve(address _approved, uint _tokenId) public {
        address owner = idToOwner[_tokenId];
        // Throws if `_tokenId` is not a valid NFT
        require(owner != address(0));
        // Throws if `_approved` is the current owner
        require(_approved != owner);
        // Check requirements
        bool senderIsOwner = (idToOwner[_tokenId] == msg.sender);
        bool senderIsApprovedForAll = (ownerToOperators[owner])[msg.sender];
>>>     require(senderIsOwner || senderIsApprovedForAll);
        // Set the approval
        idToApprovals[_tokenId] = _approved;
        emit Approval(owner, _approved, _tokenId);
    }
```

Recommendations

Consider resetting `idToApprovals` directly inside the `_burn` function instead of calling `approve`.

[M-08] Wrong supply accounting in merge function

Severity

Impact: Low

Likelihood: High

Description

The `supply` state variable shows total amount of KITTEN locked in the VotingEscrow contract.

When merging two NFTs in the VotingEscrow contract, the `from` NFT's tokens (`value0`) are transferred to the `to` NFT. However, while the NFT is burnt, the `supply` isn't reduced before calling `_deposit_for`. Then, inside `_deposit_for`, the `supply` is increased again by `value0`.

The `supply` variable will be artificially inflated by `value0` each time a merge operation occurs. This overstatement is permanent as there's no corresponding decrement. It represents an internal accounting error within the contract.

```
function merge(uint _from, uint _to) external {
        ...
        locked[_from] = LockedBalance(0, 0);
        _checkpoint(_from, _locked0, LockedBalance(0, 0));
        _burn(_from);
        _deposit_for
        //(_to, value0, end, _locked1, DepositType.MERGE_TYPE); // @audit No supply re
    }
```

In the `_deposit_for` function, the supply is always increased:

```
function _deposit_for(
        uint _tokenId,
        uint _value,
        uint unlock_time,
        LockedBalance memory locked_balance,
        DepositType deposit_type
    ) internal {
        LockedBalance memory _locked = locked_balance;
        uint supply_before = supply;

        supply = supply_before + _value; // @audit
        // original_total_supply_including_value0 + value0
        ...
    }
```

Recommendations

Decrease the `supply` in the `merge` function before calling `_deposit_for`:

```
function merge(uint _from, uint _to) external {
        ...
        locked[_from] = LockedBalance(0, 0);
        _checkpoint(_from, _locked0, LockedBalance(0, 0));
        _burn(_from);
+       supply -= value0;
        _deposit_for(_to, value0, end, _locked1, DepositType.MERGE_TYPE);
    }
```

[M-09] Desync lets bribes be received without emissions

Severity

Impact: Medium

Likelihood: Medium

Description

Because of the fact that BribeVotingRewardwill 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.

Recommendations

Allowing 1 hour post-voting window for protocols to trigger distribution.

[M-10] split() mints NFTs to msg.sender not the original owner

Severity

Impact: Medium

Likelihood: Medium

Description

VotingEscrow implementation, the split() function mints new NFTs to the msg.sender, regardless of whether msg.sender is the original owner or just an approved operator. This is inconsistent with the behavior of other functions, such as merge(), increase_amount(), and increase_unlock_time(), which preserve ownership and rely on _isApprovedOrOwner() to allow operations but does not alter the ownership.

```
function split(
        uint _from,
        uint _amount
    ) external returns (uint _tokenId1, uint _tokenId2) {
        address msgSender = msg.sender;
        require(_isApprovedOrOwner(msgSender, _from));
        //

        int128 _splitAmount = int128(uint128(_amount));
        _locked.amount = value - _splitAmount; // already checks for underflow
        // here in ^0.8.0
        _tokenId1 = _createSplitNFT(msgSender, _locked);

        _locked.amount = _splitAmount;
        _tokenId2 = _createSplitNFT(msgSender, _locked);
```

Recommendations

Modify the split() function and use the original owner of the NFT when minting new veNFTs.

[M-11] User can prevent poke() from updating voting power

Severity

Impact: Medium

Likelihood: Medium

Description

The poke() function is designed to allow anyone to update a user's voting allocation based on their current voting power. This is important because users who do not re-cast their votes (via vote()) can otherwise continue to receive rewards as if their balance never decreased. However, the system can be abused to prevent successful pokes by introducing dust-level weights in the original vote.

```
function _vote(
        uint _tokenId,
        address[] memory _poolVote,
        uint256[] memory _weights
    ) internal {
        _reset(_tokenId);
        uint _poolCnt = _poolVote.length;
        uint256 _weight = IVotingEscrow(_ve).balanceOfNFT(_tokenId);

        for (uint i = 0; i < _poolCnt; i++) {
            address _pool = _poolVote[i];
            address _gauge = gauges[_pool];
//
            if (isGauge[_gauge]) {
                uint256 _poolWeight =
                  (_weights[i] * _weight) / _totalVoteWeight;
                require(_poolWeight != 0);
```

A malicious user can:

- Cast an initial vote with one pool receiving a very small weight (e.g., 1 unit) and the rest going to others.
- Over time, as the user's voting power decays (e.g., due to the lock approaching expiration), the resulting _poolWeight for the small-weight pool rounds down to 0.
- When poke() is later called, it attempts to reapply the same voting configuration. Due to the zero-weight condition, it reverts at this check:

```
uint256 _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;
require(_poolWeight != 0,);
```

- As a result, poke() fails, and the user's used voting weight remains outdated, allowing them to earn excess bribe rewards relative to their actual, decayed voting power.

POC:

```
function testPokeFail()
        public
        returns (address pairGauge, address poolGauge)
    {
        (address pair, address pool) = testCreateGauge();

        (pairGauge, poolGauge) = (voter.gauges(pair), voter.gauges(pool));

        (uint128 gaugeFees0, uint128 gaugeFees1) = ICLPool(pool).gaugeFees();

        vm.startPrank(user1);

        // uint256 bal = veKitten.balanceOf(user1);
        uint256 veKittenId = veKitten.tokenOfOwnerByIndex(user1, 0);
        uint256 balanceWeight = veKitten.balanceOfNFT(veKittenId);

        address[] memory voteList = new address[](2);
        voteList[0] = pair;
        voteList[1] = pool;
        uint256[] memory weightList = new uint256[](2);
        weightList[0] = 1;
        weightList[1] = balanceWeight - 1;

        //@audit pool weight would be exactly 1
        // _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;
        // 1= 1*balanceWeight /(balanceWeight - 1 + 1)
        voter.vote(veKittenId, voteList, weightList);

        vm.stopPrank();

        vm.warp(block.timestamp + 2 weeks);
        //@audit weight round down to 0 and fail
        vm.expectRevert();
        voter.poke(veKittenId);
    }
```

Recommendations

Instead of reverting on zero _poolWeight, skip such pools:

```
if (_poolWeight == 0) continue;
```

[M-12] Merging or withdrawing VotingEscrow burns tokens without rewards

Severity

Impact: Medium

Likelihood: Medium

Description

VotingEscrow merge(), split(), and withdraw() functions burn the user's NFT without claiming pending rewards from the bribe system (via getReward). Since claiming bribe rewards requires ownership of the NFT (isApprovedOrOwner check), burning the NFT makes it impossible to access unclaimed rewards afterward.

In the merge() function:

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

The getReward() function that users must call to claim bribe rewards checks ownership:

`require(IVotingEscrow(_ve).isApprovedOrOwner(msg.sender, tokenId));`

Burning the NFT removes this ownership, making it impossible to later call getReward.

Recommendations

Reset token amounts instead of directly burning nft.

[M-13] Potential revert in earned() when checkpoint supply is zero

Severity

Impact: High

Likelihood: Low

Description

Inside `ExternalBribe.earned`, the reward is calculated by multiplying the checkpoint balance of the `tokenId` by `tokenRewardsPerEpoch`, then dividing by the checkpoint's total supply.

```
function earned(address token, uint tokenId) public view returns (uint) {
        uint _startTimestamp = lastEarn[token][tokenId];
        if (numCheckpoints[tokenId] == 0) {
            return 0;
        }

        uint _startIndex = getPriorBalanceIndex(tokenId, _startTimestamp);
        uint _endIndex = numCheckpoints[tokenId] - 1;

        uint reward = 0;
        // you only earn once per epoch (after it's over)
        RewardCheckpoint memory prevRewards;
        prevRewards.timestamp = _bribeStart(_startTimestamp);

        PrevData memory _prev;
        _prev._prevSupply = 1;

        if (_endIndex > 0) {
            for (uint i = _startIndex; i <= _endIndex - 1; i++) {
                _prev._prevTs = checkpoints[tokenId][i].timestamp;
                _prev._prevBal = checkpoints[tokenId][i].balanceOf;
                uint _nextEpochStart = _bribeStart(_prev._prevTs);
                // check that you've earned it
                // this won't happen until a week has passed
                if (_nextEpochStart > prevRewards.timestamp) {
                    console.log("REWARD ADDED : ");
                    console.log(prevRewards.balance);
                    reward += prevRewards.balance;
                }

                prevRewards.timestamp = _nextEpochStart;
                _prev._prevSupply = supplyCheckpoints[
                    getPriorSupplyIndex(_nextEpochStart + DURATION)
                ].supply;
                prevRewards.balance =
                    (_prev._prevBal *
                        tokenRewardsPerEpoch[token][_nextEpochStart]) /
>>>                     _prev._prevSupply;
            }
        }

        Checkpoint memory _cp0 = checkpoints[tokenId][_endIndex];
        (_prev._prevTs, _prev._prevBal) = (_cp0.timestamp, _cp0.balanceOf);

        uint _lastEpochStart = _bribeStart(_prev._prevTs);
        uint _lastEpochEnd = _lastEpochStart + DURATION;

        if (
            block.timestamp > _lastEpochEnd && _startTimestamp < _lastEpochEnd
        ) {
            SupplyCheckpoint memory _scp0 = supplyCheckpoints[
                getPriorSupplyIndex(_lastEpochEnd)
            ];
            _prev._prevSupply = _scp0.supply;

            reward += (_prev._prevBal *
                    tokenRewardsPerEpoch[token][_lastEpochStart]) /
>>>             _prev._prevSupply;
        }

        return reward;
    }
```

However, it is not considered the case where `supplyCheckpoints` at the calculated epoch is 0, it is possible in a scenario where poke is called and all

the calculated balance inside that checkpoint results in 0 supply.

Add the following test file to the codebase: https://gist.github.com/said017/f6b0674c8f992c07c69ec1c4cffa3630.

Run the test:

```
forge test --match-test testEpochDivZero -vvv
```

Recommendations

Set the `_prev._prevSupply` to 1 if it 0.

[M-14] Incorrect check inside `CLGauge.claimFees`

Severity

Impact: Medium

Likelihood: Medium

Description

Inside `CLGauge.claimFees`, it will call `pool.collectFees` and if returned `claimed0` and `claimed1` is greater than 0, it will consider it as fees.

```
function _claimFees() internal returns (uint claimed0, uint claimed1) {
        if (!isForPair) return (0, 0);
        (claimed0, claimed1) = pool.collectFees();
>>>     if (claimed0 > 0 || claimed1 > 0) {
            uint _fees0 = fees0 + claimed0;
            uint _fees1 = fees1 + claimed1;

            if (
                _fees0 > IBribe(internal_bribe).left(token0) &&
                _fees0 / ProtocolTimeLibrary.WEEK > 0
            ) {
                fees0 = 0;
                _safeApprove(token0, internal_bribe, _fees0);
                IBribe(internal_bribe).notifyRewardAmount(token0, _fees0);
            } else {
                fees0 = _fees0;
            }
            if (
                _fees1 > IBribe(internal_bribe).left(token1) &&
                _fees1 / ProtocolTimeLibrary.WEEK > 0
            ) {
                fees1 = 0;
                _safeApprove(token1, internal_bribe, _fees1);
                IBribe(internal_bribe).notifyRewardAmount(token1, _fees1);
            } else {
                fees1 = _fees1;
            }

            emit ClaimFees(msg.sender, claimed0, claimed1);
        }
    }
```

However, inside `pool`, it will set fees amount to 1 if it is empty.

```
function collectFees()
        external
        override
        lock
        onlyGauge
        returns (uint128 amount0, uint128 amount1)
    {
        amount0 = gaugeFees.token0;
        amount1 = gaugeFees.token1;
        if (amount0 > 1) {
            // ensure that the slot is not cleared, for gas savings
>>>         gaugeFees.token0 = 1;
            TransferHelper.safeTransfer(token0, msg.sender, --amount0);
        }
        if (amount1 > 1) {
            // ensure that the slot is not cleared, for gas savings
>>>         gaugeFees.token1 = 1;
            TransferHelper.safeTransfer(token1, msg.sender, --amount1);
        }

        emit CollectFees(msg.sender, amount0, amount1);
    }
```

This means if currently fees are empty, `CLGauge` will incorrectly add 1 fee to the `fees0` and `fees1`, which will eventually a cause revert due to lack of balance.

Recommendations

Adjust the following line inside `CLPool`.

```
function collectFees() external override lock onlyGauge returns
      (uint128 amount0, uint128 amount1) {
        amount0 = gaugeFees.token0;
        amount1 = gaugeFees.token1;
        if (amount0 > 1) {
            // ensure that the slot is not cleared, for gas savings
            gaugeFees.token0 = 1;
            TransferHelper.safeTransfer(token0, msg.sender, --amount0);
-         }
+         } else {
+            --amount0;
+       }
        if (amount1 > 1) {
            // ensure that the slot is not cleared, for gas savings
            gaugeFees.token1 = 1;
            TransferHelper.safeTransfer(token1, msg.sender, --amount1);
-         }
+         } else {
+            --amount1;
+       }

        emit CollectFees(msg.sender, amount0, amount1);
    }
```

[M-15] DOS attack by delegating tokens at MAX_DELEGATES = 1024

Severity

Impact: High

Likelihood: Low

Description

The delegate function in the VotingEscrow contract allows any user to delegate voting power to any other address. However, delegated token information is stored in arrays whose size is capped by MAX_DELEGATES = 1024. This limit is designed to prevent excessive gas usage during operations like transfer, withdraw, or burn.

In HyperEVM's dual-block architecture, this constraint has more severe implications:

Fast blocks: 2-second block time, 2M gas limit.

Slow blocks: 1-minute block time, 30M gas limit. A user whose NFT has received 1024 delegations will require ~23M gas to perform certain operations like transfer or withdrawal. If such an operation is attempted in a fast block, it will always fail, because the fast block's gas cap (2M) is far below the required gas. Unless the user explicitly configures their transaction to be processed in a slow block, the action will remain unconfirmed or revert indefinitely. • 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 hits the gas limit, the victim can not withdraw/transfer/delegate.

Recommendations

Reduce the MAX_DELEGATES value from 1024 to prevent bloated delegation arrays from exceeding even fast block limits.

[M-16] Double reward claim in `ExternalBribe`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `ExternalBribe` contract, a malicious user can claim rewards twice for the same `tokenId` in the same epoch due to flawed logic in the `ExternalBribe::earned()` function. This issue allows a user to receive rewards multiple times from a single epoch, which can result in other users being unable to claim their rightful rewards due to reward depletion.

The problem lies in the update of the `prevRewards.timestamp` field inside the loop. The contract updates `prevRewards.timestamp` **unconditionally**, even when `_nextEpochStart <= prevRewards.timestamp`. This causes reward accumulation logic to break, allowing rewards from a previously claimed epoch to be included again.

`ExternalBribe::earned()` function:

```
function earned(address token, uint tokenId) public view returns (uint) {
    ...
    uint _startIndex = getPriorBalanceIndex(tokenId, _startTimestamp);
    uint _endIndex = numCheckpoints[tokenId] - 1;

    uint reward = 0;
    // you only earn once per epoch (after it's over)
    RewardCheckpoint memory prevRewards;
    prevRewards.timestamp = _bribeStart(_startTimestamp);

    PrevData memory _prev;
    _prev._prevSupply = 1;

    if (_endIndex > 0) {
        for (uint i = _startIndex; i <= _endIndex - 1; i++) {
            _prev._prevTs = checkpoints[tokenId][i].timestamp;
            _prev._prevBal = checkpoints[tokenId][i].balanceOf;

            uint _nextEpochStart = _bribeStart(_prev._prevTs);
            if (_nextEpochStart > prevRewards.timestamp) {
                reward += prevRewards.balance;
            }

=>          prevRewards.timestamp = _nextEpochStart;
            _prev._prevSupply = supplyCheckpoints[getPriorSupplyIndex
              (_nextEpochStart + DURATION)].supply;

            prevRewards.balance =

                  (_prev._prevBal * tokenRewardsPerEpoch[token][_nextEpochStart]) /
                _prev._prevSupply;
        }
    }
    ...
    return reward;
}
```

An example scenario:

We'll refer to the `tokenId` as "User A".

1. **Epoch 1**:
    - User A deposits → Checkpoint 1 is created.
2. **Epoch 2**:
    - User A claims rewards → Earns Epoch 1 reward → `lastEarn` updated (green point).
3. **Epoch 2**:
    - User A makes two more deposits → Checkpoints 2 and 3 are created.

4. User A claims again:

    - `_startIndex`: Checkpoint 1.
    - `_endIndex`: Checkpoint 3.
    - `prevRewards.timestamp`: Start of Epoch 2.

    - **Iteration 1** (Checkpoint 1):

        - `_nextEpochStart = Start of Epoch 1`.
        - `prevRewards.timestamp = Start of Epoch 1`.
        - `prevRewards.balance` updated (Epoch 1 reward).

    - **Iteration 2** (Checkpoint 2):

        - `_nextEpochStart = Start of Epoch 2`.
        - Since `_nextEpochStart > prevRewards.timestamp`, `reward += Epoch 1 reward`.
        - `prevRewards.timestamp = Start of Epoch 2`.

    - **Checkpoint 3** is skipped or only used depending on current timestamp.

5. Result: User A successfully reclaimed already claimed rewards.

Recommendations

Update the logic inside `ExternalBribe::earned()` so that `prevRewards.timestamp` is only updated when `_nextEpochStart > prevRewards.timestamp`.

8.4. Low Findings

[L-01] Redundant logic in `getEpochStart()` of `ExternalBribe`

The getEpochStart function in the ExternalBribe contract contains redundant logic that can lead to unnecessary complexity.

In the `ExternalBribe` contract, the `getEpochStart` function is defined as follows:

```
function getEpochStart(uint timestamp) public pure returns (uint) {
    uint bribeStart = _bribeStart(timestamp);
    uint bribeEnd = bribeStart + DURATION;
    return timestamp < bribeEnd ? bribeStart : bribeStart + 7 days;
}
```

The issue arises because the calculation of `bribeStart + 7` days will always be greater than `timestamp` if timestamp is within the current epoch. This means that the condition checking and the alternative return value are redundant.

[L-02] No revert in `ownerOf()` for non-existent token IDs

The `ownerOf` function in the `VotingEscrow` contract does not revert when queried with a non-existent token ID. This can lead to misleading behavior in the contract.

```
function ownerOf(uint _tokenId) public view returns (address) {
    return idToOwner[_tokenId];
}
```

This is inconsistent with EIP-721:

```
/// @notice Find the owner of an NFT
    /// @dev NFTs assigned to zero address are considered invalid, and queries
    ///  about them do throw.
    /// @param _tokenId The identifier for an NFT
    /// @return The address of the owner of the NFT
    function ownerOf(uint256 _tokenId) external view returns (address);
```

[L-03] Inadequate reward validation in `ClGauge` contract

The reward rate validation in the `CLGauge` contract does not adequately account for the total balance of the kitten token, **which includes both distributable rewards and unclaimed rewards**.

```
require(rewardRate > 0);
uint balance = IERC20(kitten).balanceOf(address(this));
require(
    rewardRate <= balance / epochDurationLeft,
    "Provided reward too high"
);
```

While this check ensures that the `rewardRate` does not exceed the available balance divided by the remaining epoch duration, it fails to consider that the balance includes both rewards that are yet to be distributed and rewards that have not been claimed.

[L-04] `CLGauge` should use `ERC721HolderUpgradeable` instead of `ERC721Holder`

The `CLGauge` contract is designed to be upgradeable but currently inherits from `ERC721Holder`. It should instead inherit from `ERC721HolderUpgradeable` to ensure compatibility with upgradeable patterns.

```
contract CLGauge is
    ICLGauge,
    UUPSUpgradeable,
    Ownable2StepUpgradeable,
    ERC721Holder
{
```

Since `CLGauge` is intended to be an upgradeable contract, it is crucial to use the upgradeable version of the `ERC721Holder`, which is `ERC721HolderUpgradeable`.

[L-05] Irreversible whitelisting of tokens

The `_whitelist` function in the Voter contract allows tokens to be added to a whitelist, but once a token is whitelisted, it cannot be removed. This `one-way` operation may lead to operational inconveniences and potential misuse.

```
function _whitelist(address _token) internal {
    require(!isWhitelisted[_token]);
    isWhitelisted[_token] = true;
    emit Whitelisted(msg.sender, _token);
}
```

Once a token is added to the whitelist, there is no mechanism provided in the contract to remove it from the whitelist. This means that any token that is whitelisted remains so indefinitely, regardless of changes in the token's status or relevance to the system.

To enhance the flexibility and security of the contract, it is recommended to implement a mechanism to allow for the removal of tokens from the whitelist.

[L-06] Redundant check for `_votes` in `_reset()` function

The `_reset` function in the `Voter` contract contains a redundant check for the variable `_votes`, which is of type `uint256`.

```
uint256 _votes = votes[_tokenId][_pool];
```

The first `if` statement checks whether `_votes` is not equal to zero. However, since `_votes` is a `uint256`, it cannot be negative, and if it is not zero, it must be greater than zero. Therefore, the second check for `_votes` > 0 is redundant and leads to unreachable code in the `else` block.

```
if (_votes != 0) {
    _updateFor(gauges[_pool]);
    weights[_pool] -= _votes;
    votes[_tokenId][_pool] -= _votes;
    if (_votes > 0) {
        IBribe(internal_bribes[gauges[_pool]])._withdraw(
            uint256(_votes),
            _tokenId
        );
        IBribe(external_bribes[gauges[_pool]])._withdraw(
            uint256(_votes),
            _tokenId
        );
        _totalWeight += _votes;
    } else {
        _totalWeight -= _votes;
    }
}
```

It is recommended to remove the redundant check for `_votes > 0` and simplify the code.

[L-07] Hardcoded gas in `ClFactory`'s `getSwapFee` and `getUnstakedFee` can cause issues

In `getSwapFee` and `getUnstakedFee`, the gas provided to perform static calls to `unstakedFeeModule` and `swapFeeModule` is hardcoded.

```
    /// @inheritdoc ICLFactory
    function getSwapFee(address pool) external view override returns (uint24) {
        if (swapFeeModule != address(0)) {
            (bool success, bytes memory data) = swapFeeModule
                .excessivelySafeStaticCall(
>>>                 200_000,
                    32,
                    abi.encodeWithSelector(IFeeModule.getFee.selector, pool)
                );
            if (success) {
                uint24 fee = abi.decode(data, (uint24));
                if (fee <= 100_000) {
                    return fee;
                }
            }
        }
        return tickSpacingToFee[CLPool(pool).tickSpacing()];
    }

    /// @inheritdoc ICLFactory
    function getUnstakedFee(
        address pool
    ) external view override returns (uint24) {
        address gauge = voter.gauges(pool);
        if (!voter.isAlive(gauge) || gauge == address(0)) {
            return 0;
        }
        if (unstakedFeeModule != address(0)) {
            (bool success, bytes memory data) = unstakedFeeModule
                .excessivelySafeStaticCall(
>>>                 200_000,
                    32,
                    abi.encodeWithSelector(IFeeModule.getFee.selector, pool)
                );
            if (success) {
                uint24 fee = abi.decode(data, (uint24));
                if (fee <= 1_000_000) {
                    return fee;
                }
            }
        }
        return defaultUnstakedFee;
    }
```

Consider making it configurable to avoid potential issues in the future, in case the required gas needs to be adjusted.

[L-08] `Voter.updateForRange` could revert if `end` value is invalid

Inside `updateForRange`, the provided `end` value is used to loop over the `pools` without checking whether it exceeds the valid `pools` range`.

```
function updateForRange(uint start, uint end) public {
        for (uint i = start; i < end; i++) {
            _updateFor(gauges[pools[i]]);
        }
    }
```

This could result in a revert. Consider checking the `end` value, and if it exceeds the length of pools, use the pools length instead.

```
function updateForRange(uint start, uint end) public {
+         end = Math.min(end, pools.length);
        for (uint i = start; i < end; i++) {
            _updateFor(gauges[pools[i]]);
        }
    }
```

[L-09] Potential liquidity overflow

In the CLGauge contract, both the `deposit` and `withdraw` functions perform unsafe type casting from `uint128` to `int128` when interacting with the pool's stake function. If the liquidity value is large (exceeding 2^127-1), this conversion could result in overflow, leading to incorrect liquidity values being staked or withdrawn from the pool.

```
function deposit(
        uint256 nfpTokenId,
        uint256 tokenId
    ) public lock actionLock {
        ...
        // stake nfp liquidity in pool
        pool.stake(int128
        //(_liquidity), _tickLower, _tickUpper, true); // @audit unsafe cast to int128
        ...
    }
```

```
function withdraw(uint nfpTokenId) public lock actionLock {
        ...
        if (_liquidity != 0)
            pool.stake(-int128
            //(_liquidity), _tickLower, _tickUpper, true); // @audit unsafe cast to in
        ...
    }
```

It's recommended to use SafeCast library to cast liquidity to int128.

[L-10] Fee-on-transfer token incompatibility

The protocol does not correctly handle fee-on-transfer tokens in its reward distribution mechanisms. The current implementation assumes the full amount specified is received after transfer, which leads to accounting discrepancies when fee-on-transfer tokens are used as rewards.

The impact is that the protocol records and calculates rewards based on the pre-fee amount rather than the actual received amount. This discrepancy leads to wrong calculation of reward.

EternalBribe.notifyRewardAmount:

```
function notifyRewardAmount(address token, uint amount) external lock {
        ...

        _safeTransferFrom(token, msg.sender, address(this), amount);
        tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards + amount; //
        // @audit uses pre-fee amount
        ...
    }
```

InternalBribe.notifyRewardAmount:

```
function notifyRewardAmount(address token, uint amount) external lock {
        ...
        if (block.timestamp >= periodFinish[token]) {
            _safeTransferFrom(token, msg.sender, address
            //(this), amount); // @audit uses pre-fee amount
            rewardRate[token] = amount / DURATION;
        } else {
            uint _remaining = periodFinish[token] - block.timestamp;
            uint _left = _remaining * rewardRate[token];
            require(amount > _left);
            _safeTransferFrom(token, msg.sender, address
            //(this), amount); // @audit uses pre-fee amount
            rewardRate[token] = (amount + _left) / DURATION;
        }
        ...
    }
```

Gauge.notifyRewardAmount:

```
function notifyRewardAmount(address token, uint amount) external lock {
        ...
        if (block.timestamp >= periodFinish[token]) {
            _safeTransferFrom(token, msg.sender, address(this), amount);
            rewardRate[token] = amount / DURATION;  // @audit uses pre-fee
            // amount
        } else {
            uint _remaining = periodFinish[token] - block.timestamp;
            uint _left = _remaining * rewardRate[token];
            require(amount > _left);
            _safeTransferFrom(token, msg.sender, address(this), amount);
            rewardRate[token] =
            //(amount + _left) / DURATION; // @audit uses pre-fee amount
        }
        ...
    }
```

Recommendation: To properly handle fee-on-transfer tokens, the protocol should track the actual received amount by comparing token balances before and after transfers.

[L-11] Reward calculation reverts due to potential arithmetic overflow

The `earned` function in CLGauge.sol performs several arithmetic calculations that could potentially overflow, causing the function to revert and leading to a Denial of Service condition for reward claims:

When computing updated reward growth: `rewardGrowthGlobalX128 += (reward * FixedPoint128.Q128) / pool.stakedLiquidity()` And when calculating the final earned reward amount `(rewardGrowthInsideDelta * _liquidity) / FixedPoint128.Q128;`

In both cases, the intermediate multiplication could exceed the uint256 maximum value, causing a revert. It leads to users will be unable to claim their rewards.

```
function earned(uint256 nfpTokenId) public view returns (uint) {
        ...
        if (timeDelta != 0 && rewardReserve > 0 && pool.stakedLiquidity() > 0) {
            uint256 reward = rewardRate * timeDelta;
            if (reward > rewardReserve) reward = rewardReserve;

            rewardGrowthGlobalX128 +=

                //(reward * FixedPoint128.Q128) / // @audit revert due to overflow her
                pool.stakedLiquidity();
        }
        ...
        return
        //(rewardGrowthInsideDelta * _liquidity) / FixedPoint128.Q128; // @audit rever
    }
```

It's recommended to use OpenZeppelin Math.mulDiv that can handles intermediate overflows.

Reference: https://github.com/velodrome-finance/slipstream/blob/main/contracts/gauge/CLGauge.sol#L123. https://github.com/velodrome-finance/slipstream/blob/main/contracts/gauge/CLGauge.sol#L132.

[L-12] Initializer pattern in `ClFactory` lacks upgrade, risks frontrunning

The `CLFactory` contract uses an `initialize` function with the `initializer` modifier but does not implement any upgrade pattern (such as UUPSUpgradeable).

Recommendation to eliminate the risk of front-run initialization:

- If the contract is not intended to be upgradeable, replace the initialize function with a constructor.
- If upgradeability is desired, implement a proper upgrade pattern (e.g., UUPSUpgradeable) like the other contracts in the codebase.

[L-13] Users can create NFTs with 0 locked amount through split function

VotingEscrow doesn't allow users to create NFTs with zero locked amount.

The `split` function also doesn't allow split to 0. However, it fails to check that both resulting NFTs have non-zero amounts. When a user sets `_amount` equal to the full locked balance of the source NFT (`value`), it results in the first NFT having a zero locked amount:

```
function split(
        uint _from,
        uint _amount
    ) external returns (uint _tokenId1, uint _tokenId2) {
        ...
        // split and mint new NFTs
        int128 _splitAmount = int128(uint128(_amount));
        _locked.amount = value - _splitAmount; // @audit Can be 0 here if value
        // = _splitAmount
        _tokenId1 = _createSplitNFT(msgSender, _locked);

        _locked.amount = _splitAmount;
        _tokenId2 = _createSplitNFT(msgSender, _locked);
        ...
    }
```

This could enable users to create numerous "zombie" NFTs with zero locked amounts by repeatedly splitting their tokens, which contradicts the contract design that requires locked value for NFTs.

It's recommended to add a check in the `split` function to ensure both resulting NFTs have non-zero amounts.

```
function split(
        uint _from,
        uint _amount
    ) external returns (uint _tokenId1, uint _tokenId2) {
        ...
        int128 _splitAmount = int128(uint128(_amount));
+       require(_splitAmount < value, "Cannot split entire amount");
        _locked.amount = value - _splitAmount;
        _tokenId1 = _createSplitNFT(msgSender, _locked);
```

[L-14] Gauge may receive rewards for period it was dead when revived

When a gauge is revived using the `reviveGauge` function, its `supplyIndex` is not updated to the current index. This creates a vulnerability where a revived gauge can calculate and receive rewards for the period it was dead.

```
function reviveGauge
    //(address _gauge) external { // @audit index isn't updated
        require(msg.sender == emergencyCouncil, "not emergency council");
        require(!isAlive[_gauge], "gauge already alive");
        isAlive[_gauge] = true;
        emit GaugeRevived(_gauge);
    }
```

It's recommended to update the gauge's `supplyIndex` to the current global index.

```
function reviveGauge(address _gauge) external {
    require(msg.sender == emergencyCouncil, "not emergency council");
    require(!isAlive[_gauge], "gauge already alive");
    isAlive[_gauge] = true;
+   supplyIndex[_gauge] = index; // Update to current index
    emit GaugeRevived(_gauge);
}
```

[L-15] Rewards lost until `Gauge` or `Bribe` deposits are nonzero

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.

[L-16] Dust losses in `notifyRewardAmount`

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.

[L-17] Lack of `_disableInitializers` in upgradeable contracts

It's best practice to disable the ability to initialize the implementation contracts. The following contracts are missing `_disableInitializers()` call in their constructor:

- Voter.
- VotingEscrow.
- CLFactory.
- FactoryRegistry.
- CLGauge.

Consider adding a constructor with the OpenZeppelin `_disableInitializers` in all the upgradeable contracts:

```
constructor() {
    _disableInitializers();
}
```

[L-18] Potential out of gas risk in reward distribution checkpoint traversal

The Gauge contract generates a checkpoint for each user operation, which can lead to excessive gas consumption during reward distribution. If a user has not claimed rewards for an extended period, traversing through numerous checkpoints may result in an Out of Gas (OOG) error.

When a user claims rewards, the contract starts from the last checkpoint and iterates through all checkpoints associated with that user. If a user has not claimed rewards for a long time, the number of checkpoints can grow significantly, leading to increased gas costs during the iteration.

```
function earned(address token, address account) public view returns (uint) {
        uint _startTimestamp = Math.max(
            lastEarn[token][account],
            rewardPerTokenCheckpoints[token][0].timestamp
        );
        if (numCheckpoints[account] == 0) {
            return 0;
        }

        uint _startIndex = getPriorBalanceIndex(account, _startTimestamp);
        uint _endIndex = numCheckpoints[account] - 1;

        uint reward = 0;

        if (_endIndex > 0) {
            for (uint i = _startIndex; i <= _endIndex - 1; i++) {
                Checkpoint memory cp0 = checkpoints[account][i];
                Checkpoint memory cp1 = checkpoints[account][i + 1];
                (uint _rewardPerTokenStored0, ) = getPriorRewardPerToken(
                    token,
                    cp0.timestamp
                );
                (uint _rewardPerTokenStored1, ) = getPriorRewardPerToken(
                    token,
                    cp1.timestamp
                );
                reward +=
                    (cp0.balanceOf *
                        (_rewardPerTokenStored1 - _rewardPerTokenStored0)) /
                    PRECISION;
            }
        }

        Checkpoint memory cp = checkpoints[account][_endIndex];
        (uint _rewardPerTokenStored, ) = getPriorRewardPerToken(
            token,
            cp.timestamp
        );
        reward +=
            (cp.balanceOf *
                (rewardPerToken(token) -
                    Math.max(
                        _rewardPerTokenStored,
                        userRewardPerTokenStored[token][account]
                    ))) /
            PRECISION;

        return reward;
    }
```

This could potentially cause the transaction to run out of gas, resulting in a failed operation.

Recommendations: Allow users to claim rewards for special checkpoints.

[L-19] Inaccuracy in `Voter` contract reward distribution mechanism

The reward distribution mechanism in the `voter` contract may become inaccurate and out of sync due to its reliance on dynamic weights derived from the `IVotingEscrow(_ve).balanceOfNFT(_tokenId)`. This can lead to discrepancies in reward calculations, especially in scenarios with a high number of users.

In the `Voter` contract, the reward distribution ratio is calculated based on the total weight of votes:

```
uint256 _ratio =
//(amount * 1e18) / totalWeight; // 1e18 adjustment is removed during claim
```

The total weight is influenced by the balance of NFTs, which is determined by the following function in the `VotingEscrow` contract:

```
function _vote(
        uint _tokenId,
        address[] memory _poolVote,
        uint256[] memory _weights
    ) internal {
        _reset(_tokenId);
        uint _poolCnt = _poolVote.length;
        uint256 _weight = IVotingEscrow(_ve).balanceOfNFT(_tokenId);
        uint256 _totalVoteWeight = 0;
        uint256 _totalWeight = 0;
        uint256 _usedWeight = 0;
```

This weight can change over time. However, the reward distribution index is updated based on these weights, and the `_updateFor` function calculates shares based on the current weights.

```
function _balanceOfNFT(
        uint _tokenId,
        uint _t
    ) internal view returns (uint) {
        uint _epoch = user_point_epoch[_tokenId];
        if (_epoch == 0) {
            return 0;
        } else {
            Point memory last_point = user_point_history[_tokenId][_epoch];
            last_point.bias -=
                last_point.slope *
                int128(int256(_t) - int256(last_point.ts));
            if (last_point.bias < 0) {
                last_point.bias = 0;
            }
            return uint(int256(last_point.bias));
        }
    }
```

However, the weights are only recalculated when users call functions like `poke` or `vote`, which may not happen frequently enough to keep the weights in sync with the actual state of the system.

As a result, if many users are participating, the reward calculations may become inaccurate, leading to some users receiving more or fewer rewards than they should based on their actual contributions.

This is an unfixed issue from the forked repo(https://github.com/spearbit/portfolio/blob/master/pdfs/Velodrome-Spearbit-Security-Review.pdf 5.3.3 Bribe and fee token emissions can be gamed by users).

Recommendations: Follow the fix recommendation from the original finding. Re-designing the Voter and Bribe contracts may be worthwhile to decay user deposits automatically.

[L-20] Calling `notifyRewardAmount` for unsupported tokens can grief `gauge`

An attacker exploits the system by repeatedly calling `notifyRewardAmount` for a new `gauge`, potentially reaching the `MAX_REWARD_TOKENS` limit. This could prevent legitimate tokens from being added as rewards.

Once a token is whitelisted, it cannot be removed unless the `swapOutRewardToken` is called. This means that if the `voter` has over `MAX_REWARD_TOKENS` tokens whitelisted, an attacker could perform a griefing attack on the newly created gauge to achieve a temporary DOS.

```
function notifyRewardAmount(address token, uint amount) external lock {
        require(token != stake);
        require(amount > 0);
        if (!isReward[token]) {
            require(
                IVoter(voter).isWhitelisted(token),
                "rewards tokens must be whitelisted"
            );
            require(
                rewards.length < MAX_REWARD_TOKENS,
                "too many rewards tokens"
            );
        }
        if (rewardRate[token] == 0)
            _writeRewardPerTokenCheckpoint(token, 0, block.timestamp);
        (
            rewardPerTokenStored[token],
            lastUpdateTime[token]
        ) = _updateRewardPerToken(token, type(uint).max, true);
        _claimFees();

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

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

The attacker could provide `minimum` tokens and call `notifyRewardAmount` with a token that is no longer being rewarded (or not to be supported by the gauge). This could increase `rewards.length` by `1` since the token is whitelisted.

This could lead to the `rewards` array exceeding `MAX_REWARD_TOKENS`, preventing legitimate(real-intended) rewards tokens from being added as rewards in the future. Unless the `swapOutRewardToken` is called, a temporary DOS can't be resolved.

Recommendations: Restrict the call only to the `voter`.

[L-21] Incorrect logic and documentation in balance retrieval

The documentation and logic in the `getPriorBalanceIndex` function of the `Gauge` contract contain inaccuracies. The comments incorrectly state that the **function returns the balance as of a given block**, while **it actually returns the index based on block.timestamp**. Additionally, the logic for returning a balance of zero is flawed, leading to potential misinformation.

The relevant documentation and code snippets are as follows:

```
/**
    * @notice Determine the prior balance for an account as of a block number

        * @dev Block number must be a finalized block or else this function will reve
    * @param account The address of the account to check
    * @param timestamp The timestamp to get the balance at
    * @return The balance the account had as of the given block
    */
```

```
// Next check implicit zero balance
   if (checkpoints[account][0].timestamp > timestamp) {
       return 0;
   }
```

When the condition `checkpoints[account][0].timestamp > timestamp` is true, the function returns `0` (the index is `0`). This is misleading because returning 0 here implies that the account should have a zero balance, but it actually refers to the index of the first checkpoint.

For other cases, you could directly use `checkpoints[account][index]` to get the balance at the point, but for the case we discussed above, it might incorrectly return `checkpoints[account][0]` instead of `0`.

This is because there might be non-zero data in `checkpoints[account][0]`.

```
if (
            _nCheckPoints > 0 &&
            checkpoints[account][_nCheckPoints - 1].timestamp == _timestamp
        ) {
            checkpoints[account][_nCheckPoints - 1].balanceOf = balance;
        } else {
            checkpoints[account][_nCheckPoints] = Checkpoint(
                _timestamp,
                balance
            );
            numCheckpoints[account] = _nCheckPoints + 1;
        }
```

Additionally, things could also go wrong for `VotingEscrow::getPastVotes`.

```
function getPastVotes(
        address account,
        uint timestamp
    ) public view returns (uint) {
        uint32 _checkIndex = getPastVotesIndex(account, timestamp);
        // Sum votes
        uint[] storage _tokenIds = checkpoints[account][_checkIndex].tokenIds;
        uint votes = 0;
        for (uint i = 0; i < _tokenIds.length; i++) {
            uint tId = _tokenIds[i];
            // Use the provided input timestamp here to get the right decay
            votes = votes + _balanceOfNFT(tId, timestamp);
        }
        return votes;
    }
```

if the `timestamp` is smaller than `checkpoints[account][0].timestamp`, the incorrect voting is being calculated.

```
// Next check implicit zero balance
        if (checkpoints[account][0].timestamp > timestamp) {
            return 0;
        }
```

POC

```
//forge test -f https://rpc.hyperliquid.xyz/evm  --mt testBalanceAt -vvv
    function testBalanceAt() public {
        _setUp();
        vm.warp( 100 weeks);
        vm.startPrank(deployer);

        kitten.approve(address(veKitten), type(uint256).max);
        uint256 tokenId1=veKitten.create_lock_for
          (100_000_000 ether, 10 weeks , user1);

        vm.warp(block.timestamp + 10 weeks);
        uint256 tokenId2=veKitten.create_lock_for
          (100_000_000 ether, 10 weeks , user1);

        uint256 votingPower1= veKitten.balanceOfNFTAt
          (tokenId1, block.timestamp - 5 weeks);
        uint256 votingPower2= veKitten.balanceOfNFTAt
          (tokenId2, block.timestamp - 5 weeks);

        //@audit token2 is just created but have much more voting power at the
        // past
        assertEq(votingPower1,479452054794520547880000 );
        assertEq(votingPower2,143835616438356164364000000 );

    }
```

Recommendations

To address these issues, it is recommended to:

- Update Documentation: Revise the comments to accurately reflect that the function returns an index based on block.timestamp and clarify how to retrieve the corresponding balance.
- Set all `checkpoints` at index `0` to the empty state to accurately reflect the `0`.

[L-22] Voting restriction possible after resetting in `reset()`

In the `reset` function of the `Voter` contract, the following code is executed:

```
function reset(uint _tokenId) external onlyNewEpoch(_tokenId) {
    require(IVotingEscrow(_ve).isApprovedOrOwner(msg.sender, _tokenId));
    lastVoted[_tokenId] = block.timestamp; // Updates lastVoted
    _reset(_tokenId);
    IVotingEscrow(_ve).abstain(_tokenId);
}
```

When a user calls the `reset` function, the `lastVoted` timestamp is updated to the current `block.timestamp`. This means that the user will not be able to

`vote` in the current epoch, as the `onlyNewEpoch` modifier poses a restriction. This could lead to situations where users who intend to continue participating in voting are inadvertently locked out.

Recommendations: If the intention is to allow users to `abstain` from voting in the current epoch, then the function should be renamed from `reset` to something like `abstain`. This would better reflect the action being taken, as it would indicate that the user is choosing not to vote in the current epoch.

Otherwise, just remove the `lastVoted` assignment.

[L-23] `VotingEscrow.totalSupplyAtT` is not working properly

Inside `VotingEscrow.totalSupplyAtT`, it can be observed that it will always provide latest last epoch's `point_history` to the `_supply_at`.

```
function totalSupplyAtT(uint t) public view returns (uint) {
        uint _epoch = epoch;
        Point memory last_point = point_history[_epoch];
        return _supply_at(last_point, t);
    }
```

This means that if the provided `t` is in the past, the function will not use the correct `last_point`, which will result in an incorrect total supply calculation and cause the operation to revert.

```
function _supply_at(
        Point memory point,
        uint t
    ) internal view returns (uint) {
        Point memory last_point = point;
        uint t_i = (last_point.ts / WEEK) * WEEK;
        for (uint i = 0; i < 255; ++i) {
            t_i += WEEK;
            int128 d_slope = 0;
            if (t_i > t) {
                t_i = t;
            } else {
                d_slope = slope_changes[t_i];
            }
            last_point.bias -=
                last_point.slope *
>>>             int128(int256(t_i - last_point.ts));
            if (t_i == t) {
                break;
            }
            last_point.slope += d_slope;
            last_point.ts = t_i;
        }

        if (last_point.bias < 0) {
            last_point.bias = 0;
        }
        return uint(uint128(last_point.bias));
    }
```

PoC :

```
function testSupplyInconsistent() public {
        _setUp();

        vm.startPrank(deployer);

        kitten.approve(address(veKitten), type(uint256).max);
        veKitten.create_lock_for(100_000_000 ether, 52 weeks * 2, user1);
        veKitten.create_lock_for(150_000_000 ether, 52 weeks * 2, user2);


        vm.warp(block.timestamp + 1 weeks);

        veKitten.checkpoint();

        uint256 cacheTime = block.timestamp;
        uint256 cachetBlock = block.number;

        console.log("total supply at current block : ");
        console.log(veKitten.totalSupply());
        console.log("total supply at current block using block time : ");
        console.log(veKitten.totalSupplyAt(cachetBlock));

        vm.warp(block.timestamp + 1 weeks);

        veKitten.checkpoint();

        console.log("total supply at cached block timestamp : ");
         // will revert
        console.log(veKitten.totalSupplyAtT(cacheTime));
        console.log("total supply at cached block using block time : ");
        console.log(veKitten.totalSupplyAt(cachetBlock));


        vm.stopPrank();
    }
```

Recommendations

Use the same search logic as in `totalSupplyAt`, first determine the correct target epoch, then calculate `dt`, and pass those values to `_supply_at`.

[L-24] Inconsistent restriction on `increase_amount` and `deposit_for`

`increase_amount` and `deposit_for` perform the same operation but have inconsistent restrictions: `increase_amount` only allows the approved spender or owner of the `tokenId` to execute the operation, whereas `deposit_for` can be called by anyone.

```
function deposit_for(uint _tokenId, uint _value) external nonreentrant {
        LockedBalance memory _locked = locked[_tokenId];
        require(_value > 0); // dev: need non-zero value
        require(_locked.amount > 0, "No existing lock found");
        require(
            _locked.end > block.timestamp,
            "Cannot add to expired lock. Withdraw"
        );
        _deposit_for(
            _tokenId,
            _value,
            0,
            _locked,
            DepositType.DEPOSIT_FOR_TYPE
        );
    }
```

```
function increase_amount(uint _tokenId, uint _value) external nonreentrant {
>>>     assert(_isApprovedOrOwner(msg.sender, _tokenId));

        LockedBalance memory _locked = locked[_tokenId];

        assert(_value > 0); // dev: need non-zero value
        require(_locked.amount > 0, "No existing lock found");
        require(
            _locked.end > block.timestamp,
            "Cannot add to expired lock. Withdraw"
        );

        _deposit_for(
            _tokenId,
            _value,
            0,
            _locked,
            DepositType.INCREASE_LOCK_AMOUNT
        );
    }
```

Recommendations: Either remove the restriction on `increase_amount`, or add a restriction to `deposit_for`.

[L-25] Lack of a `_poolVote` length check inside `Voter.vote` could cause issues

User can `vote` an arbitrary amount of pools as long as it is registered inside the `Voter`.

```
function vote(
        uint tokenId,
        address[] calldata _poolVote,
        uint256[] calldata _weights
    ) external onlyNewEpoch(tokenId) {
        require(IVotingEscrow(_ve).isApprovedOrOwner(msg.sender, tokenId));
        require(_poolVote.length == _weights.length);
        lastVoted[tokenId] = block.timestamp;
        _vote(tokenId, _poolVote, _weights);
    }
```

This could cause issues. The lack of a maximum pools length check could result in an out-of-gas error, as many operations loop over the pool list and perform potentially expensive operations, such as interacting with bribe contracts.

Recommendations: Consider adding a maximum pool length check inside the `vote` operation.

[L-26] Users lose rewards if they call `getReward` exactly at epoch end

In the `ExternalBribe.earned` function, users can permanently lose rewards for an epoch if they call `getReward` exactly at the epoch's end time.

Consider a scenario when a user has one checkpoint and calls `getReward` when `block.timestamp` is exactly equal to `_lastEpochEnd`:

- The condition `block.timestamp > _lastEpochEnd` evaluates to `false`.
- The reward calculation is skipped, returning 0.
- The `getReward` function updates `lastEarn[token][tokenId]` to `block.timestamp` (which is `_lastEpochEnd`).
- When the user tries again later (for example `block.timestamp + 1`), `_startTimestamp` will be set to `lastEarn[token][tokenId]` (which equals to `_lastEpochEnd`).
- The condition `_startTimestamp < _lastEpochEnd` will now evaluate to `false`.
- The user permanently loses the rewards for that epoch.

This creates a timing vulnerability where users lose their rewards.

```
function earned(address token, uint tokenId) public view returns (uint) {
        uint _startTimestamp = lastEarn[token][tokenId];
        ...
        Checkpoint memory _cp0 = checkpoints[tokenId][_endIndex];
        (_prev._prevTs, _prev._prevBal) = (_cp0.timestamp, _cp0.balanceOf);

        uint _lastEpochStart = _bribeStart(_prev._prevTs);
        uint _lastEpochEnd = _lastEpochStart + DURATION;

        if (
            block.timestamp > _lastEpochEnd && _startTimestamp < _lastEpochEnd
            // // @audit check will not pass if user calls getReward at _lastEpochEnd
        ) {
            SupplyCheckpoint memory _scp0 = supplyCheckpoints[
                getPriorSupplyIndex(_lastEpochEnd)
            ];
            _prev._prevSupply = _scp0.supply;

            reward +=
                (_prev._prevBal *
                    tokenRewardsPerEpoch[token][_lastEpochStart]) /
                _prev._prevSupply;
        }

        return reward;
    }
```

Recommendations

Revert when users call `getReward` at epoch end to avoid losing rewards.

[L-27] Users can deposit NFTs into killed CL gauges

The Kittenswap protocol uses a gauge system to distribute rewards. For safety purposes, the protocol can "kill" gauges through the `killGauge` function in the Voter contract, which sets `isAlive[_gauge] = false` and zeroes out claimable rewards.

```
function deposit
      (uint256 nfpTokenId, uint256 tokenId) public lock actionLock {
        ...
        // attach ve
        if (tokenId > 0) {
            require(IVotingEscrow(_ve).ownerOf(tokenId) == msg.sender);
            if (tokenIds[msg.sender] == 0) {
                tokenIds[msg.sender] = tokenId;
                IVoter(voter).attachTokenToGauge(tokenId, msg.sender);
            }
            require(tokenIds[msg.sender] == tokenId);
        } else {
            tokenId = tokenIds[msg.sender];
        }

        emit Deposit(msg.sender, nfpTokenId, tokenId, _liquidity);
    }
```

However, if a user deposits an NFP token without attaching a veKITTEN token (`tokenId = 0`), the check for whether the gauge is alive is bypassed. This allows users to deposit into a gauge that has been killed, leading to unexpected behavior in CLPool fee distribution between staked and unstaked liquidity.

Recommendations: Callback to voter contract to check status of the gauge:

```
function deposit(
        uint256 nfpTokenId,
        uint256 tokenId
    ) public lock actionLock {
        ...
+       IVoter(voter).emitDeposit(tokenId, msg.sender, _liquidity);
        emit Deposit(msg.sender, nfpTokenId, tokenId, _liquidity);
    }
```

[L-28] Inaccurate reward calculation in `ExternalBribe` contract

The reward calculation in the `ExternalBribe` contract uses the supply at the end of the previous epoch while relying on the last recorded user balance from that epoch. This discrepancy can lead to inaccurate reward distributions, allowing users to exploit the system by depositing large amounts just before the epoch ends.

```
_prev._prevSupply = supplyCheckpoints[getPriorSupplyIndex
  (_nextEpochStart + DURATION)].supply;
_prev._prevBal = checkpoints[tokenId][i].balanceOf;
```

Here, `_prev._prevSupply` retrieves the supply at the end of the previous epoch, while `_prev._prevBal retrieves` the last recorded balance for the user. This means that when calculating rewards, the contract uses the supply from the end of the epoch but the user's balance from the last checkpoint, which may not accurately reflect the user's contribution during the epoch.

This design flaw allows users to deposit a significant amount of tokens just before the epoch ends, potentially receiving a disproportionate share of rewards based on the inflated supply figure, while their actual contribution during the epoch may be minimal.

Recommendations: To mitigate this issue, consider implementing a more robust reward calculation mechanism that accurately reflects user contributions throughout the entire epoch.

[L-29] `CLGauge.earned` could revert due to reward growth underflow

Inside the `earned` function, it queries the latest `getRewardGrowthInside` and then calculates the `rewardGrowthInsideDelta`.

```
function earned(uint256 nfpTokenId) public view returns (uint) {
        uint256 timeDelta = block.timestamp - pool.lastUpdated();

        uint256 rewardGrowthGlobalX128 = pool.rewardGrowthGlobalX128();
        uint256 rewardReserve = pool.rewardReserve();
        if (timeDelta != 0 && rewardReserve > 0 && pool.stakedLiquidity() > 0) {
            uint256 reward = rewardRate * timeDelta;
            if (reward > rewardReserve) reward = rewardReserve;

            rewardGrowthGlobalX128 +=
                (reward * FixedPoint128.Q128) /
                pool.stakedLiquidity();
        }
        (
            ,
            ,
            ,
            ,
            ,
            int24 _tickLower,
            int24 _tickUpper,
            uint128 _liquidity,
            ,
            ,
            ,

        ) = nfp.positions(nfpTokenId);

        uint256 rewardGrowthInsideInitial = rewardGrowthInside[nfpTokenId];
        uint256 rewardGrowthInsideCurrent = pool.getRewardGrowthInside(
            _tickLower,
            _tickUpper,
            rewardGrowthGlobalX128
        );

>>>     uint256 rewardGrowthInsideDelta = rewardGrowthInsideCurrent -
            rewardGrowthInsideInitial;
        return (rewardGrowthInsideDelta * _liquidity) / FixedPoint128.Q128;
    }
```

The calculation of the reward growth and delta implicitly relies on underflow, which will not work correctly because the current `CLGauge` uses Solidity version 0.8.

Detailed information about the root cause: https://github.com/Uniswap/v3-core/issues/573.

Recommendations: Wrap the `rewardGrowthInsideDelta` inside `unchecked` block to allow underflow.

[L-30] Inconsistent `DOMAIN_TYPEHASH`

In the `VotingEscrow::delegateBySig` function, the `domainSeparator` is computed as follows:

```
bytes32 domainSeparator = keccak256(
    abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name)), keccak256(bytes
      (version)), block.chainid, address(this))
);
```

However, the `DOMAIN_TYPEHASH` constant is defined as:

```
bytes32 public constant DOMAIN_TYPEHASH = keccak256('EIP712Domain
  (string name,uint256 chainId,address verifyingContract)');
```

Here, the `domainSeparator` includes an extra `version` field, which is not part of the defined type hash, making it incompatible with the EIP-712 standard.

Update the `DOMAIN_TYPEHASH` definition to include the `version` field, as shown below:

```
bytes32 public constant DOMAIN_TYPEHASH = keccak256('EIP712Domain
  (string name,string version,uint256 chainId,address verifyingContract)');
```