Pashov Audit Group

KittenSwap Security Review

July 31st 2025 - August 5th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out 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. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

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

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. The scope of this audit was focused on updates in Voting Escrow logic, and integration with Algebra.

5. Executive Summary

A time-boxed security review of the **Kittenswap/contracts-v2** repository was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **KittenSwap**. A total of **27** issues were uncovered.

Protocol Summary	
Project Name	KittenSwap
Protocol Type	DEX
Timeline	July 31st 2025 - August 5th 2025

Review commit hash:
- 29a49b17a2494c389861b7e18b184b515992c465 (Kittenswap/contracts-v2)

Fixes review commit hash:
- e9a0af4bf8d92ef7a0c5f61db8934e94adef9fe0 (Kittenswap/contracts-v2)

Scope

`VotingEscrow.sol` `Voter.sol` `AlgebraGaugeFactory.sol` `AlgebraGauge.sol` `AlgebraVaultFactory.sol`

6. Findings

Findings count

Severity	Amount
Critical	1
High	3
Medium	3
Low	20
Total findings	**27**

Summary of findings

ID	Title	Severity	Status
[C-01]	`carryVoteForward()` enables double voting	Critical	Resolved
[H-01]	Repeated distributions for killed gauges can block valid distributions	High	Resolved
[H-02]	Reward rates can be reset to 0 and future rewards can be stolen from voters	High	Resolved
[H-03]	Permissionless voting through `Voter::carryVoteForward()`	High	Resolved
[M-01]	Community fees wasted if no voters in `claimAndDistributeAlgebraPoolFees()`	Medium	Resolved
[M-02]	`carryVoteForward()` violates vote immutability	Medium	Resolved
[M-03]	Emissions stuck if `Voter.notifyRewardAmount()` is called without voters	Medium	Resolved
[L-01]	Some contracts not following UUPS best practices	Low	Resolved
[L-02]	Fee claiming and reward distribution continues for dead gauges	Low	Resolved
[L-03]	Rounding down in `split()` function causes dust token loss.	Low	Resolved
[L-04]	`start()` function lacks protection against multiple calls	Low	Acknowledged

ID	Title	Severity	Status
[L-05]	Missing storage `__gap` in `Voter` which is upgradeable contract	Low	Resolved
[L-06]	Missing `__UUPSUpgradeable_init()` call in `AlgebraVaultFactory` initialization	Low	Acknowledged
[L-07]	`_createGauge()` allows overwriting an existing gauge	Low	Resolved
[L-08]	Expired locks can bypass expiry restriction via `split()`, regaining voting power	Low	Resolved
[L-09]	`claimAndDistributeAlgebraPoolFees()` may revert on zero transfer failure	Low	Resolved
[L-10]	Emission dust gets stuck due to rounding in `_distribute()`	Low	Resolved
[L-11]	Changing minter before `updatePeriod()` causes missed rewards	Low	Acknowledged
[L-12]	Malicious actor and zero emissions cause DoS in `distributeAll()`	Low	Resolved
[L-13]	Lost fees in old community vault during gauge creation	Low	Acknowledged
[L-14]	`claimVotingRewardBatch()` allows early claims, missing rewards	Low	Resolved
[L-15]	Fees wrongly given to current voters for past distributions	Low	Acknowledged
[L-16]	Algebra fees distributed early lost voting rewards	Low	Acknowledged
[L-17]	`split()` function double-counts locks amounts in `supply`	Low	Resolved
[L-18]	Rewards can be distributed early with `_distribute()`	Low	Resolved
[L-19]	Emissions added before first voting epoch period remain unallocated	Low	Acknowledged
[L-20]	`split` assigns max lock time to child `NFTs`, possibly locking funds	Low	Resolved

Critical findings

[C-01] `carryVoteForward()` enables double voting

Severity

Impact: High

Likelihood: High

Description

The `Voter.carryVoteForward()` function allows voters to reuse their previous period's votes (pools and weights) for the upcoming period. However, this function does not mark the token as voted for the next period (does not set `period[nextPeriod].voted[_tokenId] = true`) :

```
function carryVoteForward(uint256 _tokenId, uint256 _fromPeriod) public {
        IVoter.Period storage ps = period[_fromPeriod];
        uint256 nextPeriod = getCurrentPeriod() + 1;

        // fetch weights from previous period
        address[] memory _poolList = ps.tokenIdVotedList[_tokenId].values();
        uint256[] memory _weightList = new uint256[](_poolList.length);

        for (uint256 i; i < _poolList.length; i++) {
            address _gauge = gauge[_poolList[i]].gauge;
            _weightList[i] = ps.tokenIdVotes[_gauge][_tokenId];
        }

        // deposit votes to next period
        _vote(nextPeriod, _tokenId, _poolList, _weightList);
    }
```

As a result, when `carryVoteForward()` is used with a token that hasn't been used to vote via `vote()` function, `checkPeriodVoted(nextPeriod)` will return `false`, which allows bypassing the `notVoted` modifier in the `VotingEscrow` contract. This modifier is used in critical functions like `merge()`, `split()`, `withdraw()`, and the ERC721-overwridden `_update()` function to ensure that tokens participating in voting cannot be manipulated or transferred mid-vote:

```
modifier notVoted(uint256 _tokenId) {
        if (
            IVoter(voter).checkPeriodVoted(
                PeriodLibrary.getPeriod(block.timestamp) + 1,
                _tokenId
            ) == true
        ) revert Voted();
        _;
    }
```

```
    function _update(
        address _to,
        uint256 _tokenId,
        address _auth
    ) internal override notVoted(_tokenId) returns (address) {
        return ERC721EnumerableUpgradeable._update(_to, _tokenId, _auth);
    }
```

For example: a malicious actor can use their NFT to vote using `carryVoteForward()` , then split this NFT (will gain multiple NFTs with larger voting power as these will be locked for the maximum locking duration), then vote again with these new NFTs in the same period (where the user has already voted for using the original NFT).

Recommendations

Update `carryVoteForward()` to set `period[nextPeriod].voted[_tokenId] = true` :

```
function carryVoteForward(uint256 _tokenId, uint256 _fromPeriod) public {
        IVoter.Period storage ps = period[_fromPeriod];
        uint256 nextPeriod = getCurrentPeriod() + 1;

        // fetch weights from previous period
        address[] memory _poolList = ps.tokenIdVotedList[_tokenId].values();
        uint256[] memory _weightList = new uint256[](_poolList.length);

        for (uint256 i; i < _poolList.length; i++) {
            address _gauge = gauge[_poolList[i]].gauge;
            _weightList[i] = ps.tokenIdVotes[_gauge][_tokenId];
        }

+       period[nextPeriod].voted[_tokenId] = true;
        // deposit votes to next period
        _vote(nextPeriod, _tokenId, _poolList, _weightList);
    }
```

High findings

[H-01] Repeated distributions for killed gauges can block valid distributions

Severity

Impact: High

Likelihood: Medium

Description

Note: Distributions for killed (standard) gauges will currently fail, so this report assumes that the killed gauge referred to is an AlgebraGauge.

When a gauge is killed, the pending emissions for that gauge, for the current period, are re-routed back to the `minter` contract. Additionally, the `es.distributed` flag is *not* set to `true`. This is done so that, when the gauge is revived, we are able to re-distribute these missed emissions:

```
    function _distribute(uint256 _period, address _gauge) internal {
...
        IVoter.Emissions storage es = ps.gaugeEmissions[_gauge];
        if (es.distributed) revert EmissionsAlreadyDistributedForPeriod();

        uint256 emissions = (ps.totalEmissions * ps.gaugeTotalVotes[_gauge]) /
            ps.globalTotalVotes;
...
        if (gauge[_pool].isAlive == false) {
            kitten.transfer(address(minter), emissions);
            emissions = 0;
        } else {
            es.amount = emissions;
            es.distributed = true;
        }

        if (gauge[_pool].isAlgebra) _claimAndDistributeAlgebraPoolFees(_pool);
        kitten.approve(_gauge, emissions);
        IGauge(_gauge).notifyRewardAmount(emissions);
...
    }
```

Because of this, an attacker is able to repeatedly trigger distributions for a killed gauge and effectively transfer emissions meant for other gauges back to the `minter` contract. This will then result in valid distribution calls failing for those other gauges, since the Voter contract will not have enough `KITTEN` tokens left to actually distribute.

It is important to note that there are two distinct impacts from this bug:

1. Valid emission distributions are temporarily blocked (affects standard *and* algebra gauges):
2. An attacker can repeatedly trigger distributions for a killed gauge, draining the Voter's KITTEN balance back to the minter contract. This prevents valid distributions for other gauges in the current period from succeeding. Those missed emissions will have to be retroactively distributed in later periods.
3. Loss of rewards for voters (only affects `AlgebraGauges`):
4. If any of the affected gauges whose distributions are blocked are `AlgebraGauges`, the pool fees they accumulated for the current period will never be claimable by voters who voted for them during that period. This happens because pool fees in `AlgebraGauges` are **period-specific**. When a distribution occurs, the fees are credited as voting rewards in the gauges `votingReward` contract, but only for the current period's voters. If the distribution for that period fails (due to the exploit), the pending fees remain in the gauge's community vault and will instead be credited as voting rewards to a later period when a distribution eventually succeeds. These voting rewards cannot be retroactively credited to the original period's voters. As a result, voters who voted for the `AlgebraGauge` in the affected period will permanently lose those rewards unless they continue voting for the same gauge in subsequent periods, until the gauge finally receives a successful distribution.

Exploit Scenario

Consider the following scenario:

We are in `period_1` and there are 3 gauges, `gauge_a`, `gauge_b`, and `gauge_c`. When voting for `period_2`, Alice allocates 45 votes to `gauge_a`, Bob allocates 45 votes to `gauge_b`, and Carol allocates 10 votes for `gauge_c`. Suppose that `gauge_c` is now killed during `period_1`. Now `period_2` begins, and suppose there are `100` KITTEN tokens that will be distributed as emissions for the gauges, so:

```
# current state before distributions

minter_contract_balance = 100
voter_contract_balance = 0

# gauge_a_expected_emissions = 45
# gauge_b_expected_emissions = 45
# gauge_c_expected_emissions = 10 (killed gauge)
```

At the beginning of `period_2`, an attacker calls `distribute(gauge_c)`, this results in the `100` KITTEN being transferred from the minter contract to the voter contract and, since `gauge_c` is killed, its 10 KITTEN emissions are routed back to the minter contract. At this point the state is:

```
# current state after distribution for gauge_c

minter_contract_balance = 10
voter_contract_balance = 90
```

Now, suppose the attacker calls `distribute(gauge_c)` again, this will result in another `10` KITTEN tokens being transferred from the voter contract back to the `minter` contract. New state:

```
# current state after second distribution call for gauge_c

minter_contract_balance = 20
voter_contract_balance = 80
```

As you can see, the attacker can call `distribute(gauge_c)` 8 more times to transfer the Voter's entire KITTEN balance back to the `minter` contract. When this happens, calls to distribute emissions for `gauge_a` and `gauge_b` will fail since the Voter contract will no longer have enough KITTEN tokens for the distributions.

An attacker is able to perform this exploit for future distributions as well, potentially blocking a majority of these future distributions. However, this can only occur for future distributions if:

1. The killed algebra gauge has not been revived.
2. The algebra pool's (associated with the killed gauge) `algebraFee` was *not* set to `0`.
3. The eternal farming virtual pool incentive for the killed gauge was *not* deactivated.

Note that deactivating a killed algebra gauge can not be undone, so this will likely not occur if the gauge is expected to be revived. If an `algebraFee` is set to `0`, then this will represent a loss for the `algebraFeeReceiver`, but this is possible. However, even if the `algebraFee` is set to `0` in the next period, the affected algebra gauges' pending fees for the current period will have been redirected to the next period, which results in a loss for users who voted for those gauges for the current period.

Technicalities for exploit: An attacker will have to ensure that their `distribute(killed_algebra_gauge)` calls get included at the top of the current period *before* any other calls to distribute to other gauges.

Recommendations

Introduce a separate flag (e.g., `es.killedDistributed`) for killed gauges. When a distribution is invoked for a killed gauge:

- If `es.killedDistributed` is already set to `true` for the given `period`, return early without transferring emissions back to the `minter` contract.
- Otherwise, perform a single distribution for the killed gauge, then set `es.killedDistributed = true`.

This ensures that a killed gauge can only trigger **one** distribution per valid period, preventing repeated distributions. This will still allow the proper emissions to be distributed retroactively when the gauge is revived.

[H-02] Reward rates can be reset to 0 and future rewards can be stolen from voters

Severity

Impact: High

Likelihood: Medium

Description

The `Voter.distribute(uint256 _period, address _gauge)` function allows callers to specify any valid past period when distributing emissions to a gauge in the current period. This is meant to allow any missed distributions for gauges to be distributed in the current period.

However, the `_distribute` function does not consider the case where the supplied `gauge` did not exist for the supplied period and therefore, `ps.gaugeTotalVotes[_gauge] == 0`. For this case, the `emissions` will be calculated as `0`, since the supplied period is a valid past period that had votes cast (so `ps.globalTotalVotes > 0`):

```
        uint256 emissions = (ps.totalEmissions * ps.gaugeTotalVotes[_gauge]) / // @audit: gauge could have 0 votes for past period
            ps.globalTotalVotes;
        address _pool = gaugeToPool[_gauge];

        // transfer emissions to minter if gauge is killed
        if (gauge[_pool].isAlive == false) {
            kitten.transfer(address(minter), emissions);
            emissions = 0;
        } else {
            es.amount = emissions;
            es.distributed = true;
        }

        if (gauge[_pool].isAlgebra) _claimAndDistributeAlgebraPoolFees(_pool);
        kitten.approve(_gauge, emissions);
        IGauge(_gauge).notifyRewardAmount(emissions);
```

This allows `_claimAndDistributeAlgebraPoolFees` and `gauge::notifyRewardAmount(0)` to be triggered *after* actual emissions for the current period have already been executed. When the gauge in question is an `AlgebraGauge`, this opens up two attack vectors:

Resetting reward rates to 0

When `algebraGauge::notifyRewardAmount(0)` is invoked, the reward rates for the associated eternal farming virtual pool will be reset to `0`:

```
// AlgebraGauge.sol

    function _notifyRewardAmount(
        uint256 _rewardAmount,
        uint256 _bonusRewardAmount
    ) internal {
        uint256 duration = PeriodLibrary.periodNext(block.timestamp) -
            block.timestamp;
...
        algebraGaugeFactory.setRates(
            key,
            SafeCast.toUint128(_rewardAmount / duration), // @audit: resets rates to `0` when
_rewardAmount == 0
            SafeCast.toUint128(_bonusRewardAmount / duration)
        );
    }
```

```
// EternalVirtualPool.sol

  function setRates(uint128 rate0, uint128 rate1) external override onlyFromFarming {
    _distributeRewards(); // @audit: distribute rewards based on previous rates
    (rewardRate0, rewardRate1) = (rate0, rate1); // @audit: update rates to 0
  }
```

An attacker can take advantage of this to block reward distribution for gauge's eternal farming virtual pool.

Suppose we are currently in `period_5` and a new `AlgebraGauge` has been added (reward rates are initialized to `0`):

```
# gauge eternal virtual pool state

reward_rate = 0
reward_reserves = 1 (1 wei of rewards added during initialization)
```

Users can now vote for this gauge for the next period, `period_6`. We now enter `period_6`, and emissions are distributed to this gauge, and the reward rates are updated:

```
# gauge eternal virtual pool state

reward_rate = x
reward_reserves = 1 + y (y is reward amount added in period_6)
```

An attacker can immediately call `distribute(4, algebraGauge)`, which will ultimately trigger `algebraGauge::notifyRewardAmount(0)` and reset the reward rates back to `0`:

```
# gauge eternal virtual pool state

reward_rate = 0
reward_reserves = 1 + y
```

In the next period, when `algebraGauge.notifyRewardAmount` is called again, the virtual pool will not distribute pending rewards since the current rate was reset to `0`.

An attacker can repeat this exploit for `n` distribution periods, where `n` is the number of valid periods that existed before the algebra gauge in question was created.

Redirecting pending rewards to the current period

Additionally, when distributions are executed, pending algebra pool fees held in the associated community vault are transferred to the `votingReward` contract for that gauge and allocated for the current period:

```
// Voter::_distribute

        if (gauge[_pool].isAlgebra) _claimAndDistributeAlgebraPoolFees(_pool);
```

```
// Voter::_claimAndDistributeAlgebraPoolFees

    function _claimAndDistributeAlgebraPoolFees(address _pool) internal {
...
        votingReward.notifyRewardAmount(address(token0), token0Claimed);
...
        votingReward.notifyRewardAmount(address(token1), token1Claimed);
        emit DistributeAlgebraFees(_pool, token0Claimed, token1Claimed);
    }
```

```
// Reward.sol

    function notifyRewardAmount(
        address _token,
        uint256 _amount
    ) public virtual nonReentrant onlyRole(NOTIFY_ROLE) {
        uint256 currentPeriod = getCurrentPeriod();
        _addReward(currentPeriod, _token, _amount);
```

Therefore, these rewards are meant for the users who voted in the previous period. An attacker is able to exploit the bug mentioned previously to trigger the `_claimAndDistributeAlgebraPoolFees` at the *end* of the current period, *after* distributions have already been made for all gauges in that period. This will result in a majority of the rewards intended for the next period being distributed for the current period.

Suppose we are in `period_5`. A new `algebraGauge` is created, and users begin to vote for it for `period_6`. During `period_5`, pool fees are being accumulated in the gauge's community vault, which further entices users to vote for this gauge. At the start of `period_6`, distributions are made for the `algebraGauge` and all pending rewards are distributed to the gauge's `votingReward` contract for `period_6`. So far, this is working as expected.

Now, during `period_6` pool fees again begin to accumulate in the vault, which entices users to again cast votes for this gauge for `period_7`. However, at the end of `period_6`, an attacker calls `distribute(4, algebraGauge)` to trigger `_claimAndDistributeAlgebraPoolFees` a second time for `period_6`. Then, when

`period_7` starts and distributions are made, less pool fees will be collected from the vault, and the voters for `period_7` will not receive their expected voting rewards. The rewards meant for `period_7` have therefore been successfully redirected to `period_6`.

Similar to the first attack vector, this exploit can be executed `n` number of times, where `n` is the number of valid past periods that occurred before the algebra gauge in question was created.

Recommendations

When distributing to gauges, if the gauge has no votes for the period, return early to avoid accidentally distributing pool fees twice in a given period and incorrectly resetting the virtual pool's reward rate to `0`.

[H-03] Permissionless voting through `Voter::carryVoteForward()`

Severity

Impact: Medium

Likelihood: High

Description

The `carryVoteForward` function in `Voter.sol` lacks essential access controls, allowing any user to manipulate votes for any NFT without authorization from the owner of it.

```
    /// @notice Carry forward votes from past period
    function carryVoteForward(uint256 _tokenId, uint256 _fromPeriod) public {
        IVoter.Period storage ps = period[_fromPeriod];
        uint256 nextPeriod = getCurrentPeriod() + 1;
```

The result of this vulnerability is that anyone can call `carryVoteForward()` for any `veKITTEN` and vote for the next period for it (by copying a previous vote of that `veKITTEN`) without this being approved by the original owner.

Recommendations

Implement access control over the `carryVoteForward()` like this :

```
        if (veKitten.isApprovedOrOwner(msg.sender, _tokenId) == false)
            revert NotApprovedOrOwner();
```

Medium findings

[M-01] Community fees wasted if no voters in `claimAndDistributeAlgebraPoolFees()`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `Voter` contract, the `_claimAndDistributeAlgebraPoolFees()` function is called during the `_distribute()` flow for Algebra gauges, where it:

1. Collects and withdraws community fees from the Algebra pool.
2. Sends these fees to the associated `VotingReward` contract via `votingReward.notifyRewardAmount()` for distribution to the gauge's voters **in the current period**.

```
function _distribute(uint256 _period, address _gauge) internal {
        //...

        uint256 emissions = (ps.totalEmissions * ps.gaugeTotalVotes[_gauge]) /
            ps.globalTotalVotes;
        //...

        // transfer emissions to minter if gauge is killed
        if (gauge[_pool].isAlive == false) {
            kitten.transfer(address(minter), emissions);
            emissions = 0;
        } else {
            es.amount = emissions;
            es.distributed = true;
        }

        if (gauge[_pool].isAlgebra) _claimAndDistributeAlgebraPoolFees(_pool); // @audit : if emissions = 0 (no voters) this will still be called to add voting rewards for a period that doesn't have voters
        //...
    }
```

```
function _claimAndDistributeAlgebraPoolFees(address _pool) internal {
        //...
        IAlgebraCommunityVault vault = IAlgebraCommunityVault(
            algebraPool.communityVault()
        );

        //...
```

```
        vault.withdraw(address(token0), token0BalVault);
        uint256 token0Claimed = token0.balanceOf(address(this)) -
            token0BalBefore;
        token0.approve(address(votingReward), token0Claimed);
        votingReward.notifyRewardAmount(address(token0), token0Claimed);

        //...
    }
```

```
// @note : VotingReward.notifyRewardAmount()
    function notifyRewardAmount(
        address _token,
        uint256 _amount
    ) public virtual nonReentrant onlyRole(NOTIFY_ROLE) {
        uint256 currentPeriod = getCurrentPeriod();
        _addReward(currentPeriod, _token, _amount);

        emit NotifyReward(currentPeriod, msg.sender, _token, _amount);
    }
```

However, if the gauge **did not receive any votes** during the current period, the withdrawn community fees are still emissioned for the current period, and since no voters on the gauge for the current period, no one is eligible to claim these rewards, resulting in these community fees (voting rewards) to be lost instead of being distributed to the voters of later periods (note: the `VotingReward` contract includes a `transferERC20()` function to rescue any stuck tokens; however, the issue here concerns the **unfair allocation** of community fees, not token recovery).

Recommendations

Before calling `_claimAndDistributeAlgebraPoolFees(_pool)`, ensure that the gauge has voters in the current period by checking:

```
function _distribute(uint256 _period, address _gauge) internal {
        //...

        uint256 emissions = (ps.totalEmissions * ps.gaugeTotalVotes[_gauge]) /
            ps.globalTotalVotes;
        //...

        // transfer emissions to minter if gauge is killed
        if (gauge[_pool].isAlive == false) {
            kitten.transfer(address(minter), emissions);
            emissions = 0;
        } else {
            es.amount = emissions;
            es.distributed = true;
        }

-       if (gauge[_pool].isAlgebra) _claimAndDistributeAlgebraPoolFees(_pool);
+       if (emissions > 0 && gauge[_pool].isAlgebra) {
+           _claimAndDistributeAlgebraPoolFees(_pool);
+       }
    }
```

[M-02] `carryVoteForward()` violates vote immutability

Severity

Impact: Medium

Likelihood: Medium

Description

The `Voter.carryVoteForward()` function allows a user to reuse their voting preferences (pools and weights) from any previous period in the next period. This is intended to simplify re-voting for users:

```
function carryVoteForward(uint256 _tokenId, uint256 _fromPeriod) public {
        IVoter.Period storage ps = period[_fromPeriod];
        uint256 nextPeriod = getCurrentPeriod() + 1;

        // fetch weights from previous period
        address[] memory _poolList = ps.tokenIdVotedList[_tokenId].values();
        uint256[] memory _weightList = new uint256[](_poolList.length);

        for (uint256 i; i < _poolList.length; i++) {
            address _gauge = gauge[_poolList[i]].gauge;
            _weightList[i] = ps.tokenIdVotes[_gauge][_tokenId];
        }

        // deposit votes to next period
        _vote(nextPeriod, _tokenId, _poolList, _weightList);
    }
```

However, the protocol enforces that votes for a given token ID can only be used to vote once per period. This invariant is respected in the `vote()` function by checking if the token has already voted in the target period (via `_checkPeriodVoted()`):

```
function vote(
        uint256 _tokenId,
        address[] memory _poolList,
        uint256[] memory weightList
    ) public {
        uint256 nextPeriod = getCurrentPeriod() + 1;

        // do all the checks
        if (_checkPeriodVoted(nextPeriod, _tokenId))
            revert AlreadyVotedInPeriod();
        //...
    }
```

The `carryVoteForward()` function, however, lacks such a check. If the token has already voted in the next period, calling `carryVoteForward()` will overwrite the existing vote, thus breaking the invariant that vote changes can only happen once per period.

Recommendations

Add a check in `carryVoteForward()` to ensure that the token has not already voted in the next period, implementing the same logic in the `vote()` function.

[M-03] Emissions stuck if `Voter.notifyRewardAmount()` is called without voters

Severity

Impact: Medium

Likelihood: Medium

Description

In the `Voter` contract, the `notifyRewardAmount()` function allows anyone to donate `KITTEN` tokens for the current period, which are distributed among registered gauges based on their share of the total votes. This function is also invoked by `Minter.updatePeriod()` to mint and allocate emissions starting from the next epoch period after the `Voter.start()` call.

However, the function does not validate whether the current period has any active voters; it does not check if `period[_period].globalTotalVotes > 0`. As a result, if the function is called during a period with no votes, the donated emissions will be stuck in the contract and will not be distributed to any gauge.

```
  function notifyRewardAmount(uint256 _amount) public {
        uint256 currentPeriod = getCurrentPeriod();

        kitten.safeTransferFrom(msg.sender, address(this), _amount);
        period[currentPeriod].totalEmissions += _amount;
    }
```

```
function _distribute(uint256 _period, address _gauge) internal {
        //...
        IVoter.Period storage ps = period[_period];
        IVoter.Emissions storage es = ps.gaugeEmissions[_gauge];
        if (es.distributed) revert EmissionsAlreadyDistributedForPeriod();

        uint256 emissions = (ps.totalEmissions * ps.gaugeTotalVotes[_gauge]) /
            ps.globalTotalVotes;
       //...
    }
```

Recommendations

Disallow calling `Voter.notifyRewardAmount()` if the current period has no voters by checking `period[_period].globalTotalVotes == 0`, or alternatively, track undistributed rewards and carry them over to the next period.

Low findings

[L-01] Some contracts not following UUPS best practices

The `AlgebraVaultFactory` and `AlgebraGauge` are both deployed behind a UUPS proxy, but do not call `_disableInitializers` inside of their `constructors`. This means that anyone can call `initialize` directly on the implementation contracts to become the owner of the implementations. Note that this seemingly poses no current risk since the main attack vector that comes from this is from exposing delegatecall functionality in the implementation that can then trigger a `selfdestruct`. However, the implementations do not have `delegatecall` functionality, and `selfdestruct` no longer deletes a contract's code.

That being said, in the spirit of following best security practices, consider calling `_disableInitializers` inside of the contracts' constructors.

[L-02] Fee claiming and reward distribution continues for dead gauges

The `_distribute()` function processes fee claiming for dead gauges. Even when a gauge is killed (meaning the `isAlive` is false, the distribution logic still executes fee claiming from Algebra pools :

```
        // transfer emissions to minter if gauge is killed
        if (gauge[_pool].isAlive == false) {
            kitten.transfer(address(minter), emissions);
            emissions = 0;
        } else {
            es.amount = emissions;
            es.distributed = true;
        }

        if (gauge[_pool].isAlgebra) _claimAndDistributeAlgebraPoolFees(_pool); // <<<------
```

As a result Algebra pool fees are claimed and processed for dead gauges while the kitten emissions are not distributed since the `isAlive` is false.

Consider not claiming the algebra pool fees to veKITTEN gauge voters, if the gauge is killed.

[L-03] Rounding down in `split()` function causes dust token loss.

The `split()` function uses integer division, which rounds down, potentially causing permanent loss of small token amounts:

```
        for (uint256 i; i < _weightList.length; i++) {
            // calc amount based on weight
            uint256 newAmount = (lockedAmount * _weightList[i]) / totalWeight;
```

The result of this is that small amounts of tokens are permanently lost due to rounding, accumulating over time across multiple split operations.

Consider allocating any remaining amount to the last NFT to prevent dust loss.

[L-04] `start()` function lacks protection against multiple calls

The `start()` function can be called multiple times without any protection in order not to be called after the actual start of the Voter :

```
    /// @notice Start ve33 voting and epoch 0
    function start() external onlyOwnerOrAuthorized {
        epoch0Period = getCurrentPeriod();
        minter.start();
    }
```

This can cause a serious issue since `minter.start` call is updating the `lastMintedPeriod = getCurrentPeriod();`. So if `start()` gets called and KITTEN mint has not yet been done for the current period of `Voter`, this will effectively result in the KITTEN emissions for the whole period not being distributed.

Consider adding this safe check in order to prevent multiple calls to the `start()` :

```
require(epoch0Period == 0, "Already started");
```

[L-05] Missing storage `__gap` in `Voter` which is upgradeable contract

The `Voter` contract inherits from multiple OpenZeppelin upgradeable contracts but lacks a storage gap (`uint256[50] private __gap;`) to reserve slots for future upgrades, potentially causing storage layout conflicts.

```
contract Voter is
    IVoter,
    UUPSUpgradeable,
    Ownable2StepUpgradeable,
    AccessControlUpgradeable,
    ReentrancyGuardUpgradeable
{
    using SafeERC20 for IERC20;
    using EnumerableSet for EnumerableSet.AddressSet;
```

Add a storage gap at the end of the contract:

```
uint256[50] private __gap;
```

[L-06] Missing `__UUPSUpgradeable_init()` call in `AlgebraVaultFactory` initialization

The `AlgebraVaultFactory.initialize()` function inherits from `UUPSUpgradeable` but does not call `__UUPSUpgradeable_init()` during initialization:

```
  function initialize(
      address _algebraFactory,
      address _voter,
      address _initialOwner
  ) public initializer {
      __Ownable2Step_init();
      __Ownable_init(_initialOwner);

      algebraFactory = _algebraFactory;
      voter = _voter;
  }
```

While `UUPSUpgradeable` in recent versions may not require explicit initialization, the inconsistent pattern compared to other initializer calls in the other contracts of the protocol, creates potential confusion.

Consider adding the missing call :

```
__UUPSUpgradeable_init();
```

[L-07] `_createGauge()` allows overwriting an existing gauge

In the `Voter` contract, the `_createGauge()` function can be called by the contract owner or any authorized address to create a new Algebra gauge or a standard gauge for a given pool.

If the gauge to be deployed is of a standard type (not Algebra), there is no check to prevent overwriting an existing gauge for the same pool:

```
function _createGauge(
      address _poolAddress,
      bool _isAlgebra
  ) internal returns (address _gauge, address _votingReward) {
      Gauge storage gs = gauge[_poolAddress];

      //...
      gs.votingReward = votingRewardFactory.createVotingReward();
      //...

      if (gs.isAlgebra) {
          // create new AlgebraCommunityVault for gauge and set to pool
          gs.vault = algebraVaultFactory.createVaultForPool(
              _poolAddress,
              msg.sender,
              address(0),
              _token0,
              _token1
```

```
            );

            //...
            gs.gauge = algebraGaugeFactory.createGauge(
                _poolAddress,
                seedAmount
            );
        } else {
            gs.gauge = gaugeFactory.createGauge(_poolAddress, gs.votingReward);
        }
       //...
    }
```

Recommendation:
Add a check to ensure that a gauge for the given pool has not already been created before allowing `_createGauge()` to proceed.

[L-08] Expired locks can bypass expiry restriction via `split()`, regaining voting power

In the `VotingEscrow` contract, the `increase_unlock_time()` function is designed to allow lock owners to extend the lock duration, thereby increasing their voting power. However, it includes an important check that disallows expired locks from extending their unlock time, as it requires that the lock's current unlock time is greater than `block.timestamp`:

```
function increase_unlock_time(
        uint256 _tokenId,
        uint256 _lock_duration
    ) external nonReentrant onlyAuthorized(_tokenId) {
        LockedBalance memory _locked = locked[_tokenId];
        //...

        if (block.timestamp >= _locked.end) revert LockExpired();
        //...
    }
```

Despite this, the restriction can be bypassed via the `split()` function, where an expired lock can be split into multiple new locks (NFTs) using user-defined weights, and each newly created lock is automatically granted the maximum lock duration (two years), **regardless of whether the original lock was expired**.

Since `split()` does not check whether the original lock has expired, it allows users to effectively reset an expired lock's duration and restore voting power, which violates the intended logic of time-based voting decay and may result in unfair voting power.

Recommendation:
Introduce an expiry check in the `split()` function to ensure it cannot be called on expired locks.

[L-09] `claimAndDistributeAlgebraPoolFees()` may revert on zero transfer failure

In the `Voter` contract, the `_claimAndDistributeAlgebraPoolFees()` function is invoked during `_distribute()` for Algebra gauges. It performs the following:

1. Collects and withdraws accumulated fees (`token0` , `token1`) from the Algebra pool's community vault.
2. Transfers these tokens to the `VotingReward` contract via `votingReward.notifyRewardAmount()` , which pulls them from the `Voter` .

```
function _claimAndDistributeAlgebraPoolFees(address _pool) internal {
       IAlgebraPool algebraPool = IAlgebraPool(_pool);
       IVotingReward votingReward = IVotingReward(gauge[_pool].votingReward);
       IERC20 token0 = IERC20(algebraPool.token0());
       IERC20 token1 = IERC20(algebraPool.token1());
       IAlgebraCommunityVault vault = IAlgebraCommunityVault(
           algebraPool.communityVault()
       );

       uint256 token0BalVault = token0.balanceOf(address(vault));
       uint256 token0BalBefore = token0.balanceOf(address(this));
       vault.withdraw(address(token0), token0BalVault);
       uint256 token0Claimed = token0.balanceOf(address(this)) -
           token0BalBefore;
       token0.approve(address(votingReward), token0Claimed);
       votingReward.notifyRewardAmount(address(token0), token0Claimed);

      //...
   }
```

If either of the collected tokens (e.g., BNB) reverts on a **zero-value transfer** (when the `VotingReward` contract tries to pull the reward amount), the call to `VotingReward.notifyRewardAmount()` will revert if the balance is zero. This causes `_distribute()` to fail, resulting in stuck emissions for the corresponding gauge.

This scenario can occur if no trading activity has taken place yet for the pool, and the community vault has not accumulated any fees, leaving its balance at zero.

Recommendation: Before proceeding with the reward distribution, add checks to ensure the claimed fee amounts (`token0Claimed` , `token1Claimed`) are greater than zero.

[L-10] Emission dust gets stuck due to rounding in `_distribute()`

In the `Voter` contract, the `_distribute()` function calculates the emission share (reward) for each registered gauge based on the proportion of votes it received relative to the total votes in a given period.

However, the reward calculation will be rounded down due to integer division. These unallocated dust emissions will accumulate in the `Voter` contract and become permanently stuck, as these cannot be claimed or withdrawn.

```
function _distribute(uint256 _period, address _gauge) internal {
      //...
      uint256 emissions = (ps.totalEmissions * ps.gaugeTotalVotes[_gauge]) /
          ps.globalTotalVotes;
      //...
  }
```

Recommendation: Track the unallocated dust emissions and allocate them for future periods.

[L-11] Changing minter before `updatePeriod()` causes missed rewards

The `Voter.setMinter()` function allows an authorized role to update the `minter` address. This minter is responsible for calling `notifyRewardAmount()` to emit rewards for each period.

```
function setMinter(address _minter) external onlyOwnerOrAuthorized {
      minter = IMinter(_minter);
  }
```

However, if the `minter` is changed **before** the previous minter calls `updatePeriod()` to finalize and emit the current period's emissions, those emissions may be missed. This is because the new minter will only start emitting rewards beginning from the **next period** (i.e., `current period + 1`). As a result, rewards for the current (or even previous) periods might be lost if distribution hasn't occurred.

```
  function updatePeriod() external returns (bool) {
      uint256 currentPeriod = getCurrentPeriod();
      if (currentPeriod > lastMintedPeriod) {
         //...
          kitten.approve(address(voter), emissions);
          voter.notifyRewardAmount(emissions);
          //...
      }
      return false;
  }
```

Recommendation: Add a check or mitigation mechanism to ensure that `updatePeriod()` has been called before allowing the minter address to be changed via `setMinter()`.

[L-12] Malicious actor and zero emissions cause DoS in `distributeAll()`

In the `Voter` contract, there are several functions responsible for distributing `KITTEN` rewards to registered gauges for the current period:

- `distributeAll()`: collects fees and distributes rewards to all registered gauges.
- `distributeRange()`: distributes rewards to a subset of registered gauges.
- `distribute(address _gauge)`: distributes rewards to a specific gauge.

All these functions delegate reward calculation to `_distribute()`, which checks whether `gaugeEmissions[_gauge].distributed == false` to prevent double distribution. If the rewards have already been distributed, `_distribute()` reverts with `EmissionsAlreadyDistributedForPeriod`:

```
function _distribute(uint256 _period, address _gauge) internal {
        // check to see if trying to distribute for future period
        if (_period > getCurrentPeriod()) revert InvalidPeriod();

        IVoter.Period storage ps = period[_period];
        IVoter.Emissions storage es = ps.gaugeEmissions[_gauge];
        if (es.distributed) revert EmissionsAlreadyDistributedForPeriod(); //! @audit
        //...
        kitten.approve(_gauge, emissions);
        IGauge(_gauge).notifyRewardAmount(emissions); //! @audit
        //...
    }
```

This opens the door for a malicious actor to call `distribute(_gauge)` on specific gauges ahead of a `distributeAll()` or `distributeRange()` call. Since those calls include the same gauge in their iteration and `_distribute()` will revert for already-distributed gauges, the entire `distributeAll()` or `distributeRange()` transaction will revert.

In addition, if the `emissions` calculated for a normal (non-Algebra) gauge is zero, the call to `IGauge(_gauge).notifyRewardAmount(emissions)` will result in a `rewardRate` of zero, which will cause the call to revert:

```
 function notifyRewardAmount(
        uint256 _amount
    ) external onlyVoterOrAuthorized updateReward(address(0)) {
        _claimFees();

        kitten.safeTransferFrom(msg.sender, address(this), _amount);

        if (block.timestamp >= finishAt) {
            rewardRate = (_amount * PRECISION) / DURATION; //! @audit : if emissions = 0,
reardRate will be zero
        } else {
            uint256 remainingRewards = (finishAt - block.timestamp) *
                rewardRate;
            if (_amount * PRECISION <= remainingRewards)
```

```
                revert NotifyLessThanEqualToLeft();
            rewardRate = ((_amount * PRECISION) + remainingRewards) / DURATION;
        }
        if (rewardRate == 0) revert ZeroRewardRate();
        //...
    }
```

Recommendation: In both `distributeAll()` and `distributeRange()` functions, skip calling `_distribute()` for gauges that have already received their rewards by checking `gaugeEmissions[_gauge].distributed == true`, and update the `_distribute()` function to check that `emissions > 0` before calling `IGauge(_gauge).notifyRewardAmount(emissions)`:

```
function _distribute(uint256 _period, address _gauge) internal {
        //...
+       if (emissiosn > 0){
        kitten.approve(_gauge, emissions);
        IGauge(_gauge).notifyRewardAmount(emissions);
+       }
        //...
    }
```

[L-13] Lost fees in old community vault during gauge creation

When `createAlgebraGauge()` creates a new community vault and switches the pool to use it, any fees accumulated in the old vault become permanently trapped and inaccessible.

```
// replace old with new vault
algebraPool.setCommunityVault(gs.vault);
```

The impact of this vulnerability is that the old community vault of the `AlgebraPool` would be replaced by the new one but the fees accumulated in the old one are not handled or recovered.

It is recommended to add a fee recovery before switching vaults.

```
// Check and migrate any existing fees
if (oldVault.balanceOf(token0) > 0 || oldVault.balanceOf(token1) > 0) {
    // Withdraw existing fees from old vault
    oldVault.withdraw(token0, oldVault.balanceOf(token0));
    oldVault.withdraw(token1, oldVault.balanceOf(token1));
    // Transfer to new vault or distribute immediately
}
```

[L-14] `claimVotingRewardBatch()` allows early claims, missing rewards

In the `Voter` contract, the `claimVotingRewardBatch()` function enables voters to claim their accumulated voting rewards from the associated `VotingReward` contracts for the gauges they have voted on:

```
function claimVotingRewardBatch(
        address[] memory _votingRewardList,
        uint256 _tokenId
    ) external {
        if (veKitten.isApprovedOrOwner(msg.sender, _tokenId) == false)
            revert NotApprovedOrOwner();

        for (uint256 i; i < _votingRewardList.length; i++)
            IVotingReward(_votingRewardList[i]).getRewardForOwner(_tokenId);
    }
```

However, this function allows users to claim rewards **up to and including the current period**, even though:

- Rewards for the current period **may still be accumulating**, via `notifyRewardAmount()` calls.
- Once a user claims their reward for the current period, the `VotingReward` contract **marks the period as claimed** for that user.

```
function getRewardForOwner(uint256 _tokenId) external virtual nonReentrant {
        if (msg.sender != address(voter)) revert NotVoter();

        _getRewardForTokenId(_tokenId, veKitten.ownerOf(_tokenId));
    }
```

```
function _getRewardForTokenId(
        uint256 _tokenId,
        address _to
    ) internal virtual {
        uint256 len = rewardTokenList.length();
        address[] memory tokenList = rewardTokenList.values();
        uint256 currentPeriod = getCurrentPeriod();
        for (uint256 i; i < len; ) {
            uint256 period = fullClaimedPeriod[_tokenId] > periodInit
                ? fullClaimedPeriod[_tokenId]
                : periodInit;
            for (; period <= currentPeriod; ) { //! @audit : allows claiming for the current
period, where rewards are not finalized
                _getReward(period, _tokenId, tokenList[i], _to);
                unchecked {
                    ++period;
                }
            }
            unchecked {
                ++i;
            }
        }
        fullClaimedPeriod[_tokenId] = currentPeriod; //! @audit : this marks the period as
claimed, so the user can't claim again for that period
    }
```

This means that:

- Any additional rewards sent to the gauge's `VotingReward` contract **after the user claims** will **not be claimable** by the user, even though they had a legitimate voting weight.
- This leads to **missed rewards** for users who claim early in the period.
- It also **incentivizes strategic claiming** where users attempt to predict the best moment to claim, defeating the purpose of fair distribution.

Recommendations

Restrict `claimVotingRewardBatch()` so that it only allows claims up to `currentPeriod - 1`, ensuring users do not miss out on any rewards still being collected during the current period.

[L-15] Fees wrongly given to current voters for past distributions

In the `Voter` contract, the `_claimAndDistributeAlgebraPoolFees()` function is used to:

1. Collect community fees from the Algebra pool via `AlgebraCommunityVault.collectFees()`.
2. Forward these fees to the `VotingReward` contract via `votingReward.notifyRewardAmount()`, which pulls the tokens from the `Voter` and distributes them to voters of the corresponding gauge **for the current period**:

```
function _distribute(uint256 _period, address _gauge) internal {
      // check to see if trying to distribute for future period
      if (_period > getCurrentPeriod()) revert InvalidPeriod();
      //...
      if (es.distributed) revert EmissionsAlreadyDistributedForPeriod();
      //...
      if (gauge[_pool].isAlgebra) _claimAndDistributeAlgebraPoolFees(_pool);
      //...
  }
```

```
function _claimAndDistributeAlgebraPoolFees(address _pool) internal {
      //...
      IAlgebraCommunityVault vault = IAlgebraCommunityVault(
          algebraPool.communityVault()
      );

      //...
      vault.withdraw(address(token0), token0BalVault);
      uint256 token0Claimed = token0.balanceOf(address(this)) -
          token0BalBefore;
      token0.approve(address(votingReward), token0Claimed);
      votingReward.notifyRewardAmount(address(token0), token0Claimed);

      //...
  }
```

```
// @note : VotingReward.notifyRewardAmount()
    function notifyRewardAmount(
        address _token,
        uint256 _amount
    ) public virtual nonReentrant onlyRole(NOTIFY_ROLE) {
        uint256 currentPeriod = getCurrentPeriod();
        _addReward(currentPeriod, _token, _amount);

        emit NotifyReward(currentPeriod, msg.sender, _token, _amount);
    }
```

This logic works correctly when `_distribute()` is called in the current period. However, there is a subtle but impactful issue when rewards have **not yet been distributed for earlier periods**.

Since `distribute(uint256 _period, address _gauge)` allows retroactive distribution of rewards for any past period up to the current one, a call to distribute a **previous period's** rewards will still:

- Pull **all** accumulated fees up to the **current time** from the community vault.
- Distribute those fees to voters **of the current period**, not the targeted `_period`.

As a result, voters who voted in the **previous period** (i.e., `_period < current period`) **will not receive any portion** of the community fees that accrued during their voting period, instead, the **current period's** voters receive all the community fees, which leads to mis-attribution and unfair distribution of rewards.

Recommendations

Implement a mechanism that separates community fee accrual by period, or track fee balances per period, to ensure that the correct amount of community fees is attributed and distributed only to voters of the appropriate period.

[L-16] Algebra fees distributed early lost voting rewards

In the `Voter` contract, the `_claimAndDistributeAlgebraPoolFees()` function is responsible for collecting accumulated community fees from an Algebra pool and distributing them to voters via the `VotingReward` contract. This function is invoked within the `_distribute()` function, which handles the reward distribution process for a specific gauge in a specific period.

However, there is a subtle issue: `_distribute()` can be called at any point during the **current period** (a 7-day window), and only **once per gauge per period**. When `_claimAndDistributeAlgebraPoolFees()` is triggered early in the period, only the **community fees collected up to that point** are pulled from the `AlgebraCommunityVault` and forwarded to the `VotingReward`. Any **additional fees** accumulated **later in the same period** are **left undistributed for the current period**, as a second `_distribute()` call is not allowed for the same gauge in that period.

This results in **lost rewards for voters**, since a portion of the community fees will not be distributed to them for the period they voted in.

```
function _distribute(uint256 _period, address _gauge) internal {
      // check to see if trying to distribute for future period
      if (_period > getCurrentPeriod()) revert InvalidPeriod();
      //...
      if (es.distributed) revert EmissionsAlreadyDistributedForPeriod();
      //...
      if (gauge[_pool].isAlgebra) _claimAndDistributeAlgebraPoolFees(_pool);
      //...
  }
```

```
function _claimAndDistributeAlgebraPoolFees(address _pool) internal {
      //...
      IAlgebraCommunityVault vault = IAlgebraCommunityVault(
          algebraPool.communityVault()
      );

      //...
      vault.withdraw(address(token0), token0BalVault);
      uint256 token0Claimed = token0.balanceOf(address(this)) -
          token0BalBefore;
      token0.approve(address(votingReward), token0Claimed);
      votingReward.notifyRewardAmount(address(token0), token0Claimed);

      //...
  }
```

```
// @note : VotingReward.notifyRewardAmount()
    function notifyRewardAmount(
        address _token,
        uint256 _amount
    ) public virtual nonReentrant onlyRole(NOTIFY_ROLE) {
        uint256 currentPeriod = getCurrentPeriod();
        _addReward(currentPeriod, _token, _amount);

        emit NotifyReward(currentPeriod, msg.sender, _token, _amount);
    }
```

Recommendations

Delay the `_distribute()` call until the **start of the next period** to ensure that all community fees accumulated for the previous period have been finalized and distributed to gauge voters.

[L-17] `split()` function double-counts locks amounts in `supply`

In the `VotingEscrow` contract, the `split()` function allows an NFT lock owner to divide their locked amount into multiple NFTs based on a user-specified weight distribution. Each newly created NFT lock receives the maximum lock duration (two years), where this function works as a reallocation of an existing locked amount.

However, the function incorrectly **inflates the** `supply`. While the original lock's amount is not deducted from the `supply` during the split process, the locked amounts for the new locks are **added** again via calls to `deposit_for()`. This results in the original lock amount being **counted twice** in the `supply`:

```
function split(
        uint256 _from,
        uint256[] memory _weightList
    )
        external
        nonReentrant
        onlyAuthorized(_from)
        notVoted(_from)
        returns (uint256[] memory tokenIdList)
    {
        //...
        for (uint256 i; i < _weightList.length; i++) {
            //...
            _deposit_for(
                tokenId,
                newAmount,
                maxUnlockTime,
                locked[tokenId],
                DepositType.MERGE_TYPE
            );
        }
        //...
    }
```

```
function _deposit_for(
        uint256 _tokenId,
        uint256 _value,
        uint256 unlock_time,
        LockedBalance memory locked_balance,
        DepositType deposit_type
    ) internal {
        LockedBalance memory _locked = locked_balance;
        uint256 supply_before = supply;

        supply = supply_before + _value;
       //...
    }
```

This leads to an incorrect `supply` value emitted in events and observable on-chain, which can:

- Mislead external integrators relying on the `supply` for calculations.
- Cause incorrect behavior in other contracts or off-chain systems interacting with this contract.
- Result in falsely inflated metrics for total locked tokens.

Recommendations

Before redistributing the locked amount via `deposit_for()` in `split()`, ensure the original lock's amount is subtracted from the `supply`.

[L-18] Rewards can be distributed early with `_distribute()`

In the `Voter` contract, the `notifyRewardAmount()` function **allows anyone to donate** `KITTEN` **tokens** to be distributed to registered gauges based on their share of total votes in the current period. This function is also called by `Minter.updatePeriod()` to mint and allocate emissions starting from the next epoch after `Voter.start()`:

```
function notifyRewardAmount(uint256 _amount) public {
      uint256 currentPeriod = getCurrentPeriod();

      kitten.safeTransferFrom(msg.sender, address(this), _amount);
      period[currentPeriod].totalEmissions += _amount;
  }
```

However, **since reward distribution is allowed in the same period, the reward is notified**, a problem arises when `notifyRewardAmount()` is called more than once in the same period:

- The first call allows `distributeAll()` to distribute the emissions, after which `gaugeEmissions[_gauge].distributed` is set to `true` for all processed gauges.
- If a second call to `notifyRewardAmount()` is made within the same period, the newly donated rewards will remain stuck in the contract as `_distribute()` will revert for gauges whose `distributed` flag is already set.

```
function _distribute(uint256 _period, address _gauge) internal {
        // check to see if trying to distribute for future period
        if (_period > getCurrentPeriod()) revert InvalidPeriod();

        IVoter.Period storage ps = period[_period];
        IVoter.Emissions storage es = ps.gaugeEmissions[_gauge];
        if (es.distributed) revert EmissionsAlreadyDistributedForPeriod();

        uint256 emissions = (ps.totalEmissions * ps.gaugeTotalVotes[_gauge]) /
            ps.globalTotalVotes;
       //...
    }
```

Recommendations

Update the `_distribute()` function to reject reward distributions for the current period, enforcing that rewards can only be distributed starting from the next period after they are notified:

```
function _distribute(uint256 _period, address _gauge) internal {
-       // check to see if trying to distribute for future period
-       if (_period > getCurrentPeriod()) revert InvalidPeriod();

+       // check to see if trying to distribute for future or current period
+       if (_period >= getCurrentPeriod()) revert InvalidPeriod();

        IVoter.Period storage ps = period[_period];
        IVoter.Emissions storage es = ps.gaugeEmissions[_gauge];
        if (es.distributed) revert EmissionsAlreadyDistributedForPeriod();
```

```
        uint256 emissions = (ps.totalEmissions * ps.gaugeTotalVotes[_gauge]) /
            ps.globalTotalVotes;
        //...
    }
```

[L-19] Emissions added before first voting epoch period remain unallocated

The `Voter.notifyRewardAmount()` function allows anyone to donate `KITTEN` tokens to be distributed across gauges proportionally to their vote share for the current period. This same function is also used by `Minter.updatePeriod()` to mint emissions into the `Voter` contract starting from `epoch0Period + 1` period.

However, there is a scenario where emissions can become stuck: if `notifyRewardAmount()` is called while the current period is still within epoch0 period (`epoch0Period == 0`) where the `Voter.start()` hasn't been called yet, then the donated emissions are not distributed, since `vote()` can only be called to vote in the next period (epoch0 period + 1), then when `distributeAll()` is called to distribute the received emissions to the gauges, it will revert as **there is no recorded votes for any of the gauges in epoch0 period**, so this will result in these emissions to be stuck in the `Voter` contract and never allocated to any gauge.

```
  function notifyRewardAmount(uint256 _amount) public {
        uint256 currentPeriod = getCurrentPeriod();

        kitten.safeTransferFrom(msg.sender, address(this), _amount);
        period[currentPeriod].totalEmissions += _amount;
    }
```

```
function vote(
        uint256 _tokenId,
        address[] memory _poolList,
        uint256[] memory weightList
    ) public {
        uint256 nextPeriod = getCurrentPeriod() + 1;
        //...
        _vote(nextPeriod, _tokenId, _poolList, weightList);
    }
```

```
    function start() external onlyOwnerOrAuthorized {
        epoch0Period = getCurrentPeriod();
        minter.start();
    }
```

Recommendations

Prevent calls to `notifyRewardAmount()` if the owner hasn't called `Voter.start()` function yet, or alternatively, track undistributed rewards and carry them over to the first valid period.

[L-20] `split` assigns max lock time to child `NFTs`, possibly locking funds

When splitting an NFT, the code sets the unlock time of each child NFT to the maximum lock time, even if the parent NFT has an earlier unlock time.

```
        uint256 maxUnlockTime = ((block.timestamp + MAXTIME) / WEEK) * WEEK;
        ...
            _mint(ownerOfFrom, tokenId);
            _deposit_for(
                tokenId,
                newAmount,
                maxUnlockTime,
```

This can unintentionally force the user into the longest possible lock duration, regardless of their original lock period. In systems where users can manually extend their lock duration at will (via `increase_unlock_time`), automatically assigning a max unlock time during `split` is unnecessary and can go against the user's actual intent.

For example: - A user locks for **3 months**, intending to unlock soon. - They decide to split the NFT to diversify votes across gauges. - The resulting child NFTs now have a **2 year** max lock time. - The user is now forced to wait the full 2 years to withdraw their locked tokens, even though their original NFT would have unlocked much sooner.

Recommendations

Consider having the child NFTs inherit the unlock time of the parent NFT during `split`. Note that the unlock time would also have to be validated to ensure that the parent NFT's lock has not already expired.