KittenSwap Security Review

Pashov Audit Group

Conducted by: Shaka, btk, 0x37, DemoreXTess

June 12th 2025 - June 21st 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 - 65c8bdd6df9cc63cc2654a7329cee8591a79770a

fixes review commit hash - 83798b6c99f61c87ac800e92624cf363a649ee22

Scope

The following smart contracts were in scope of the audit:

- `CLGauge`
- `CLGaugeFactory`
- `Gauge`
- `GaugeFactory`
- `RebaseReward`
- `Reward`
- `VotingReward`
- `VotingRewardFactory`
- `Kitten`
- `Minter`
- `Voter`
- `VotingEscrow`

7. Executive Summary

Over the course of the security review, Shaka, btk, 0x37, DemoreXTess engaged with KittenSwap to review KittenSwap. In this period of time a total of **19** issues were uncovered.

Protocol Summary

Protocol Name	KittenSwap
Repository	https://github.com/Kittenswap/contracts
Date	June 12th 2025 - June 21st 2025
Protocol Type	DEX

Findings Count

Severity	**Amount**
Critical	3
High	1
Medium	3
Low	12
Total Findings	**19**

Summary of Findings

ID	Title	Severity	Status
[C-01]	RebaseReward fails because of incorrect token handling	Critical	Resolved
[C-02]	Lack of lastMintedPeriod update allows unlimited minting of Kitten	Critical	Resolved
[C-03]	Incentive rewards may be stolen	Critical	Resolved
[H-01]	votingReward not set on Gauge	High	Resolved
[M-01]	Rewards from RebaseReward unclaimable after veKITTEN changes	Medium	Resolved
[M-02]	Accrued fees become stuck when gauge is killed	Medium	Resolved
[M-03]	Voters' rewards may be manipulated	Medium	Acknowledged
[L-01]	Missing token recovery function in Gauge, VotingReward and RebaseReward	Low	Resolved
[L-02]	Unsafe transfer in CLGauge.transferERC20()	Low	Resolved
[L-03]	Missing setter for rebase rate in Minter	Low	Resolved
[L-04]	Mismatch emission plan between doc and implementation	Low	Acknowledged
[L-05]	Inconsistent implementation for rewardRate	Low	Acknowledged
[L-06]	Missing rollover mechanism for Gauge	Low	Acknowledged
[L-07]	Gauge still can be used after killing it	Low	Resolved

[L-08]	Gauge is not updated before killing	Low	Resolved
[L-09]	Gauge rewards not distributed when total supply is zero	Low	Resolved
[L-10]	Rewards may become locked in Voter	Low	Resolved
[L-11]	Missing gap in Reward contract	Low	Resolved
[L-12]	User cannot claim reward if his lock is expired	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] `RebaseReward` fails because of incorrect token handling

Severity

Impact: High

Likelihood: High

Description

`RebaseReward` deposits rewards earned by a token ID into the `VotingEscrow` contract.

```
if (reward > 0) {
                veKitten.deposit_for(_tokenId, reward);
                emit ClaimReward(_period, _tokenId, _token, _owner);
        }
```

While the `notifyRewardAmount()` function is overridden to only allow rewards for the Kitten token, `incentivize()` still allows anyone to send rewards of any whitelisted token.

As a result, rewards of tokens other than Kitten will incorrectly deposit Kitten in the `VotingEscrow` contract.

This will cause some users to receive the Kitten rewards that correspond to other users, while the token rewards deposited with the `incentivize()` function will get locked in the contract.

Proof of concept

Add the following code to the file `TestRebaseReward.t.sol` and run `forge test -f https://rpc.hyperliquid.xyz/evm --mt test_audit_RebaseRewardBrokenRewards --mc TestRebaseReward`.

```
function test_audit_RebaseRewardBrokenRewards() public {
        test_CreateGauge();
        CLGauge _gauge = CLGauge(gauge.get(poolList[0]));
        RebaseReward _rebaseReward = RebaseReward(voter.rebaseReward());
        address user1 = userList[0];
        address user2 = userList[1];
        uint256 kittenAmount = 1 ether;

        vm.startPrank(address(voter));
        // user1 votes
        uint256 tokenId1 = veKitten.tokenOfOwnerByIndex(user1, 0);
        _rebaseReward._deposit(1 ether, tokenId1);

        // user2 votes
        uint256 tokenId2 = veKitten.tokenOfOwnerByIndex(user2, 0);
        _rebaseReward._deposit(1 ether, tokenId2);
        vm.stopPrank();

        // user1 (or anyone else) sends 1e18 token0 as reward to RebaseReward
        deal(address(_gauge.token0()), user1, 1 ether);
        vm.startPrank(address(user1));
        _gauge.token0().approve(address(_rebaseReward), 1 ether);
        _rebaseReward.incentivize(address(_gauge.token0()), 1 ether);
        vm.stopPrank();

        // notify reward after epoch ends/star of next epoch
        vm.warp(ProtocolTimeLibrary.epochNext(block.timestamp));
        vm.startPrank(address(minter));
        kitten.mint(address(minter), kittenAmount);
        kitten.approve(address(_rebaseReward), kittenAmount);
        _rebaseReward.notifyRewardAmount(address(kitten), kittenAmount);
        vm.stopPrank();

        // user1 claims reward
        (int128 veKittenBalBefore, ) = veKitten.locked(tokenId1);
        vm.prank(user1);
        _rebaseReward.getRewardForTokenId(tokenId1);
        (int128 veKittenBalAfter, ) = veKitten.locked(tokenId1);

        // all Kitten rewards have been claimed by user1
        uint256 user1Claimed = uint256(uint128(veKittenBalAfter - veKittenBalBefore));
        assertEq(user1Claimed, kittenAmount);

        // user2 tries to claim reward, but it reverts, as there is no Kitten left
        vm.prank(user2);
        vm.expectRevert();
        _rebaseReward.getRewardForTokenId(tokenId2);
}
```

Recommendations

Override the `incentivize()` function in `RebaseReward` to allow only rewards for Kitten.

Another option would be accepting rewards of any token and adapting `_getReward()` to handle the rewards for these tokens correctly.

[C-02] Lack of `lastMintedPeriod` update allows unlimited minting of Kitten

Severity

Impact: High

Likelihood: High

Description

`Minter.updatePeriod()` is meant to mint Kitten tokens for the `RebaseReward` and `Voter` contracts once per period. However, this function misses updating the `lastMintedPeriod` variable. As a result, once the first period has passed, the following condition will always be true:

```
if (currentPeriod > lastMintedPeriod) {
```

This will allow minting Kitten tokens indefinitely, leading to an unlimited supply of Kitten tokens.

Proof of concept

Add the following code to the file `TestVoter.t.sol` and run `forge test -f https://rpc.hyperliquid.xyz/evm --mc TestVoter --mt test_audit_unlimitedMinting`.

```
function test_audit_unlimitedMinting() public {
        test_Vote();
        vm.warp(block.timestamp + 1 weeks);

        uint256 totalSupplyBefore = kitten.totalSupply();
        for (uint256 i; i < 10; i++) {
            minter.updatePeriod();
            uint256 totalSupplyAfter = kitten.totalSupply();
            assertGt(totalSupplyAfter, totalSupplyBefore);
            totalSupplyBefore = totalSupplyAfter;
        }
    }
```

Recommendations

```
emit Mint(
                msg.sender,
                emissions,
                circulatingSupply(),
                _tailEmissions
            );
+
+           lastMintedPeriod = currentPeriod;

            return true;
```

[C-03] Incentive rewards may be stolen

Severity

Impact: High

Likelihood: High

Description

In VotingReward, voters can gain some rewards because of their vote. There are two kind of possible rewards in VotingReward.

1. Kitten token from voter.
2. Users can add some incentive tokens via function `incentivize`.

If users incentive some rewards via function `incentivize`, we will record these rewards in the next period. Users can vote for the related pool to get some rewards.

The problem here is that we miss one `_period` validation in function `getRewardForPeriod`. Users can get the future period's rewards.

Let's consider below scenario:

1. Alice incentives 500 USDC for next period, next period = X.
2. Bob as the first voter, votes for the related pool.
3. Bob gets rewards for the period X immediately. Currently, Bob is the only person who vote for this pool in this period. Bob will get all the rewards.

```
function incentivize(
        address _token,
        uint256 _amount
    ) external virtual nonReentrant {
        // Here we have one white list for token. So we cannot manipulate the
        // token.
        if (voter.isWhitelisted(_token) == false)
            revert NotWhitelistedRewardToken();
        uint256 currentPeriod = getCurrentPeriod() + 1;
        uint256 amount = _addReward(currentPeriod, _token, _amount);
    }
    function _deposit(uint256 _amount, uint256 _tokenId) external onlyVoter {
        uint256 nextPeriod = getCurrentPeriod() + 1;

        tokenIdVotesInPeriod[nextPeriod][_tokenId] += _amount;
        totalVotesInPeriod[nextPeriod] += _amount;
    }
    function getRewardForPeriod(
        uint256 _period,
        uint256 _tokenId,
        address _token
    ) external nonReentrant {
        // Only owner or approved can get rewards.
        if (!veKitten.isApprovedOrOwner(msg.sender, _tokenId))
            revert NotApprovedOrOwner();
        _getReward(_period, _tokenId, _token, msg.sender);
    }
```

Recommendations

Add one input parameter check, don't allow to get a reward from one future period.

8.2. High Findings

[H-01] `votingReward` not set on `Gauge`

Severity

Impact: Medium

Likelihood: High

Description

The `Gauge` contract does not set the `votingReward` during its initialization and does not have a setter function for it.

As a result, when `notifyRewardAmount()` is called and fees are claimed from the pair, the transaction will revert.

```
(claimed0, claimed1) = IPair(address(lpToken)).claimFees();
        (address _token0, address _token1) = IPair(address(lpToken)).tokens();
        if (claimed0 > 0) {
            IERC20(_token0).approve(address(votingReward), claimed0);
    @>      votingReward.notifyRewardAmount(_token0, claimed0);
```

Until a new version of the `Gauge` is deployed, the fees will not be distributed and the `notifyRewardAmount()` function will always revert, potentially causing an unfair distribution of rewards once the new `Gauge` implementation is deployed, as the distribution among depositors might have changed.

Recommendations

Set `votingReward` on `Gauge` initialization.

8.3. Medium Findings

[M-01] Rewards from `RebaseReward` unclaimable after `veKITTEN` changes

Severity

Impact: High

Likelihood: Low

Description

Voting for pools generates rewards in `VotingReward` and `RebaseReward` for the next period. This means that when a vote is reset, there might be rewards pending from the current and previous periods.

If `withdraw()`, `merge()`, or `split()` is called on a `veKITTEN` token, `locked[_tokenId]` gets `amount` and `end` set to 0, but the token is not burned. This will allow the owner of the token to claim rewards from `VotingReward`, but not from `RebaseReward`, as this contract executes the `VotingEscrow.deposit_for()` function, which requires a non-zero amount locked and the end timestamp not to be reached.

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

        if (_value == 0) revert Invalid();
@>      if (_locked.amount == 0) revert LockNotExist();
@>      if (block.timestamp >= _locked.end) revert LockExpired();
```

As a result, the owner of the token will not be able to claim the pending rewards from `RebaseReward`.

Recommendations

Create a function in `VotingEscrow` that allows reusing an existing token ID to create a new lock, so that `deposit_for()` can be called without reverting.

[M-02] Accrued fees become stuck when gauge is killed

Severity

Impact: High

Likelihood: Low

Description

The internal `_claimFees()` function, responsible for claiming these fees, is only invoked during reward notifications via `notifyRewardAmount()`. However, once the gauge is killed, new rewards are redirected to the minter, and `notifyRewardAmount()` is no longer called for the gauge:

```
claimable[_gauge] = 0;
        if (_claimable > 0) kitten.transfer(minter, _claimable);
```

```
if (isAlive[_gauge]) {
                    claimable[_gauge] += _share;
                } else {
                    IERC20(kitten).transfer(minter, _share);
                }
```

As a result, any fees that have accumulated remain stuck in the pair contract and cannot be claimed, leading to a loss of value for users.

```
// Only called within notifyRewardAmount()
function _claimFees() internal { ... }
```

Recommendations

Introduce a public function, callable only by the `voter`, that invokes `_claimFees()`. Ensure this function is called when the gauge is killed, allowing any remaining fees to be claimed and distributed appropriately.

[M-03] Voters' rewards may be manipulated

Severity

Impact: Low

Likelihood: High

Description

In voter contract, voters vote for different gauges. Voters can gain some rewards from the rebaseReward and votingReward contract according to the actual voting weight. If the voting weight is higher, the voter may gain more rewards.

The problem here is that veToken's voting power will decrease over time. When the voter finishes voting at the start of the new voting period. Malicious users can trigger `poke` function when we are near to the end of the voting period to update voting weight. Then users may get fewer rewards than expected.

When one veToken has one long locking period, voting power may not decrease too much in 7 days. However, if one veToken's locking period is expired in one month, malicious users can cause that users may lose 1/3 or 1/2 of rewards in this reward epoch period.

```
function _vote(
        uint256 _tokenId,
        address[] memory _poolVote,
        uint256[] memory _weights
    ) internal {
        IVotingReward(votingReward[_gauge])._deposit(
            uint256(_poolWeight),
            _tokenId
        );
        rebaseReward._deposit(uint256(_poolWeight), _tokenId);
    }
    function poke(uint256 _tokenId) external {
        address[] memory _poolVote = poolVote[_tokenId];
        uint256 _poolCnt = _poolVote.length;
        uint256[] memory _weights = new uint256[](_poolCnt);

        for (uint256 i = 0; i < _poolCnt; i++) {
            _weights[i] = votes[_tokenId][_poolVote[i]];
        }

        _vote(_tokenId, _poolVote, _weights);
    }
```

Recommendations

`poke` function aims to help users to update their voting power. The problem here is that if the voter has already voted, malicious users can still poke to manipulate the reward a little bit.

8.4. Low Findings

[L-01] Missing token recovery function in `Gauge`, `VotingReward` and `RebaseReward`

While the `CLGauge` contract has a token recovery function, the `Gauge`, `VotingReward`, and `RebaseReward` contracts do not.

This prevents the recovery of rewards not distributed, dust amounts resulting from rounding errors, or tokens sent by mistake to these contracts.

It is recommended to implement a token recovery function in these contracts.

[L-02] Unsafe transfer in `CLGauge.transferERC20()`

The `CLGauge.transferERC20()` function is meant to rescue tokens from the gauge, however, this might not be possible for all tokens.

Some tokens do not implement the ERC20 standard properly but are still widely used. This includes tokens that do not return a boolean value from the `transfer()` and `transferFrom()` functions. When these sorts of tokens are cast to `IERC20`, their function signatures do not match, and, therefore, the calls made revert.

It is recommended to use the `safeTransfer()` function from OpenZeppelin's `SafeERC20` library.

[L-03] Missing setter for rebase rate in `Minter`

The `Minter` contract has a `MAX_REBASE_RATE` constant. Additionally, on initialization of the `rebaseRate` variable, there is a comment that says "30% bootstrap". This suggests that the rebase rate is intended to be adjustable, but there is no setter function provided to change it after deployment.

If the rebase rate is meant to be adjustable, it should have a setter function that allows the owner to change it.

[L-04] Mismatch emission plan between doc and implementation

According to the doc(https://docs.kittenswap.finance/tokenomics/initial-distribution), our initial emission will start from `20,000,000 KITTEN (2%)`.

When we check our implementation, the actual initial amount is `20,000,000 + 20,000,000 * 30% + (20,000,000 * 130% * 5%)`. This will be larger than the expected emission amount in our doc.

```
function updatePeriod() external returns (bool) {
        uint256 currentPeriod = getCurrentPeriod();
        if (currentPeriod > lastMintedPeriod) {
            uint256 emissions = nextEmissions;
            nextEmissions = calculateNextEmissions(emissions);
            // 2% * circulating supply
            uint256 _tailEmissions = tailEmissions();
            uint256 rebase = calculateRebase(emissions);
            // emssion + emission * 30%
            uint256 treasuryEmissions = ((emissions + rebase) * treasuryRate) /
                PRECISION;
            uint256 mintAmount = emissions + rebase + treasuryEmissions;
            // kitten token in this minter.
            uint256 kittenBal = kitten.balanceOf(address(this));
            if (kittenBal < mintAmount) {
                kitten.mint(address(this), mintAmount - kittenBal);
            }
        }
    }
```

Recommendation: Consistent between the doc and the implementation.

[L-05] Inconsistent implementation for `rewardRate`

We have two different kinds of gauges according to different pools. When we notify rewards, the reward rate's calculation way is different.

In gauge, we will use `(_amount * PRECISION) / DURATION`. In CLGauge, we will use `rewardRate = amount / epochDurationLeft;`. The `epochDurationLeft` may be less than 1 week, depending on the different timestamp.

In voter, we have one function `distro`. It's possible that all gauges' rewards will be distributed in one transaction. This will cause that in CL Gauges, in the next 7 days, we will distribute rewards in one time slot, and in another time slot, we will not distribute any reward.

```
function notifyRewardAmount(
        uint256 _amount
    ) external onlyVoterOrAuthorized updateReward(address(0)) {
        _claimFees();
        // reword token is kitten.
        kitten.safeTransferFrom(msg.sender, address(this), _amount);
        // If last period is finished, we will start one new period.
        if (block.timestamp >= finishAt) {
            // PRECISION = 1e18
            rewardRate = (_amount * PRECISION) / DURATION;
        }
    }
    function notifyRewardAmount(uint256 amount) external nonReentrant {
        require(amount > 0);
        require(msg.sender == address(voter));
        _claimFees();
        // When we notify rewards, we need to update or accrue the previous
        // rewards.
        pool.updateRewardsGrowthGlobal();
        address msgSender = msg.sender;
        uint256 timestamp = block.timestamp;
        // epoch remaining time.
        // epochDurationLeft may be less than 1 week.
        uint256 epochDurationLeft = ProtocolTimeLibrary.epochNext(timestamp) -
            timestamp;
        if (block.timestamp >= periodFinish) {
            rewardRate = amount / epochDurationLeft;
            pool.syncReward({
                rewardRate: rewardRate,
                rewardReserve: amount,
                periodFinish: nextPeriodFinish
            });
        }
    }
```

Recommendation: Suggest using a similar formula for the rewardRate calculation.

[L-06] Missing rollover mechanism for Gauge

CLGauge supports rollover mechanism. For instance, if the reward period is passed inactively. Rewards are forwarded for the next notified reward amount.

```
// CLGauge
        address msgSender = msg.sender;
        uint256 timestamp = block.timestamp;
        uint256 epochDurationLeft = ProtocolTimeLibrary.epochNext(timestamp) -
            timestamp;

        kitten.safeTransferFrom(msgSender, address(this), amount);
 @>     amount = amount + pool.rollover();

        uint256 nextPeriodFinish = timestamp + epochDurationLeft;
```

This mechanism does not exist in Gauge implementation and the inactive rewarding period's reward is locked in the contract.

[L-07] Gauge still can be used after killing it

CLGauge always checks is it whether killed or not. But this is not the case for gauge. Users still can use gauge.

```
// Gauge
    function deposit(
        uint256 _amount
    ) external nonReentrant actionLock updateReward(msg.sender) {
        if (_amount == 0) revert ZeroAmount();
        lpToken.safeTransferFrom(msg.sender, address(this), _amount);
        balanceOf[msg.sender] += _amount;
        totalSupply += _amount;
    }
...
// CLGauge
    function deposit(uint256 nfpTokenId) public nonReentrant actionLock {
        if (
            IVoter(voter).isGauge(address(this)) == false ||
            IVoter(voter).isAlive(address(this)) == false
        ) revert NotGaugeOrNotAlive();
```

Consider implementing the same logic inside of the gauge.

[L-08] Gauge is not updated before killing

```
function killGauge(address _gauge) external onlyRole(AUTHORIZED_ROLE) {
        if (isAlive[_gauge] == false) revert GaugeDead();
        isAlive[_gauge] = false;
@>        uint256 _claimable = claimable[_gauge];
        claimable[_gauge] = 0;
        if (_claimable > 0) kitten.transfer(minter, _claimable);
        emit GaugeKilled(_gauge);
    }
```

Claimable value is out-dated in a given case. Less token will be sent to minter and gauge tokens will be stuck in the `Voter` contract. Instead, update the gauge reward before killing it.

[L-09] `Gauge` rewards not distributed when total supply is zero

`Gauge` updates rewards whenever a new action is performed (deposit, withdraw, claim rewards, or notify rewards). If the `totalSupply` is zero, the `rewardPerToken` will not be updated, while the last updated timestamp will be updated.

```
modifier updateReward(address _account) {
        rewardPerTokenStored = rewardPerToken();
        updatedAt = lastTimeRewardApplicable();

(...)

    function rewardPerToken() public view returns (uint256) {
        if (totalSupply == 0) {
            return rewardPerTokenStored;
        }
```

This will cause that for periods where the total supply is zero, the rewards to be distributed will get locked. While there are economic incentives for at least one user to have a non-zero balance in the gauge, this is not guaranteed, especially at the moment of creation of the gauge or after reviving it, as users may not deposit LP tokens immediately.

Proof of concept

Add the following code to the file `TestGauge.t.sol` and run `forge test -f https://rpc.hyperliquid.xyz/evm --mt test_audit_GaugeLockedRewards --mc TestGauge`.

```
function test_audit_GaugeLockedRewards() public {
        // setup
        test_CreateGauge();
        Gauge _gauge = Gauge(gauge.get(pairListVolatile[0]));
        address user1 = userList[0];
        address lpToken = address(_gauge.lpToken());
        vm.prank(kitten.minter());
        kitten.mint(address(voter), 2 ether);
        deal(lpToken, user1, 1 ether);

        // voter distributes 1e18 KITTEN to the gauge
        vm.prank(address(voter));
        _gauge.notifyRewardAmount(1 ether);

        // after 1 week, user1 deposits LP tokens into the gauge
        vm.warp(block.timestamp + 1 weeks);
        vm.startPrank(user1);
        IERC20(lpToken).approve(address(_gauge), 1 ether);
        _gauge.deposit(1 ether);
        vm.stopPrank();

        // voter distributes another 1e18 KITTEN to the gauge
        vm.prank(address(voter));
        _gauge.notifyRewardAmount(1 ether);

        // after another week, user1 claims rewards
        vm.warp(block.timestamp + 1 weeks);
        vm.prank(user1);
        _gauge.getReward(user1);

        // gauge has 1e18 KITTEN balance, but no rewards left
        assertGe(kitten.balanceOf(address(_gauge)), 1 ether);
        assertEq(_gauge.left(), 0);
}
```

Recommendations Handle the distribution or recovery of rewards not distributed due to the total supply being zero. This can be done in different ways, such as:

- Sending the non-distributed rewards to the VotingReward contract.
- Sending the non-distributed rewards to a specific address designated for this purpose.
- Adding the non-distributed rewards to the next distribution cycle.

[L-10] Rewards may become locked in `Voter`

When a gauge is killed, its claimable rewards are transferred to the minter, and subsequent rewards are redirected there as well. However, if the gauge is later revived before `updateGauge()` is called, and an index update (via the public `notifyRewardAmount()` function) occurs just before that, rewards intended for the minter will become locked in the `Voter` contract.

This is because the `supplyIndex` is updated, but the gauge's state is not, preventing the proper distribution of these rewards.

```
// reviveGauge() does not call updateGauge(), leading to locked rewards
function reviveGauge(address gauge) external onlyOwner {
    // missing: updateGauge(gauge);
    // ... revive logic ...
}
```

Recommendations

Modify the `reviveGauge()` function to call `updateGauge()` for the revived gauge.

[L-11] Missing gap in Reward contract

Kittenswap contracts are upgradeable contracts that may require gaps for future updates but only Reward contract is used as parent contracts. Voting reward and Rebase reward contracts use Reward contract as parents. However, required gap is missing in Reward contract and it may cause storage collusion due to a new upgrade on Reward contract.

For now, Voting Reward contract is safe because there is no storage variable inside it but Rebase Reward contract is not safe for Reward contract upgrade because `kitten` variable is used in the storage.

```
contract RebaseReward is IRebaseReward, Reward {
    IERC20 public kitten;
```

If any new variable is added to Reward, kitten storage variable will be overridden by it due to a missing gap in Reward contract.

Recommendations

Consider adding gap into Reward contract for storage updates.

[L-12] User cannot claim reward if his lock is expired

In Rebase Reward contract, rewards are distributed to voting escrow locks as additional lock amount. Whenever a user claims reward `deposit_for` function is called in `_getReward` function.

```
function _getReward(
        uint256 _period,
        uint256 _tokenId,
        address _token,
        address _owner
    ) internal override {
        if (totalVotesInPeriod[_period] > 0) {
            uint256 reward = _earned(
              _period,
              _tokenId,
              _token
            ); reward is not kitten, directly transfer it
            tokenIdRewardClaimedInPeriod[_period][_tokenId][_token] += reward;

            if (reward > 0) {
@>              veKitten.deposit_for(_tokenId, reward);
                emit ClaimReward(_period, _tokenId, _token, _owner);
            }
        }
    }
```

Voting escrow doesn't allow depositing into expired locks.

Scenario:

1. User locked 100e18 tokens into voting escrow for 2 years.
2. User voted pools and rebase reward generated yield for the user.
3. 2 years later user wanted to claim his rewards but it's not possible.

Therefore, users who don't claim rewards actively will lose all of their rewards due to design.

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

        if (_value == 0) revert Invalid();
        if (_locked.amount == 0) revert LockNotExist();
@>      if (block.timestamp >= _locked.end) revert LockExpired()
```

Recommendations

Consider another way to deposit these rewards into voting escrow such as a permissioned function which can be called only by rebase reward.