Etherex Contracts Security Review

Auditors

Desmond Ho, Lead Security Researcher

Hyh, Lead Security Researcher

M4rio.eth, Lead Security Researcher

Anurag Jain, Security Researcher

T1moh, Associate Security Researcher

Report prepared by: Lucas Goiriz

October 23, 2025

Contents

1 About Spearbit **2**

2 Introduction **2**

3 Risk classification **2**
3.1 Impact . . . 2
3.2 Likelihood . . . 2
3.3 Action required for severity levels . . . 2

4 Executive Summary **3**

5 Findings **4**
5.1 Low Risk . . . 4
5.1.1 Incorrect fee amount set when gauge is killed . . . 4
5.1.2 `distribute()` will revert for killed CL gauges . . . 4
5.1.3 Edge case: Temporary DoS . . . 5
5.1.4 `updateFeeDistributorForGauge()` doesn't update some mappings . . . 6
5.1.5 Incorrect state variable used for flash gross fee accounting . . . 6
5.1.6 It's possible to withdraw from `VoteModule` bypassing `unlockTime` . . . 6
5.1.7 It's impossible to remove vulnerable `nfpManager` without AccessHub upgrade . . . 9
5.1.8 Revived gauges can have stale redirection that will receive emissions instead of them . . . 10
5.1.9 Emission update is incorrect . . . 11
5.1.10 `xRex` token is not added to `rewards` in `GaugeV3.sol` . . . 11
5.1.11 Emission rewards are stuck when there is no liquidity in pool . . . 12
5.1.12 `RamsesV3PositionManager`'s liquidity operations can be blocked after period flips . . . 15
5.1.13 `RamsesV3PositionManager` can excessively slash user . . . 16
5.1.14 CL gauges can have orphaned `feeDistributor` . . . 17
5.2 Gas Optimization . . . 18
5.2.1 `lastClaimByToken` can be updated to `currentPeriod` to avoid extra iteration . . . 18
5.2.2 Cache `r33()` and consolidate sync functions . . . 18
5.2.3 Redundant status in `WhitelistRevoked` event . . . 18
5.2.4 Redundant `augmentGaugeRewardsForPair()` function . . . 19
5.2.5 Redundant `ClGaugeFactoryStorage` library . . . 19
5.2.6 Unrequired condition . . . 19
5.2.7 `lastClaimByToken` is not updated correctly . . . 19
5.2.8 `voter` type can be changed to `IVoter` . . . 20
5.2.9 `positionLastModified` can be merged and tightly packed into `Position` struct . . . 20
5.2.10 Position key computation can be performed in `assembly` . . . 23
5.3 Informational . . . 24
5.3.1 Incorrect comment . . . 24
5.3.2 Incorrect errors . . . 24
5.3.3 Incorrect fee amount logged . . . 24
5.3.4 `Abstained` event isn't meaningful . . . 25
5.3.5 `GaugeV3.getPeriodReward()` is not called by position manager . . . 25
5.3.6 `GaugeV3.getPeriodReward()` should not update `lastClaimByToken` . . . 25
5.3.7 Function `Voter.vote()` can be simplified . . . 27
5.3.8 `feeRecipient` should be directly fetched from `pool` . . . 27
5.3.9 Improper Access Control in `FeeCollector` and `AccessHub` interaction . . . 27
5.3.10 Unused Variable . . . 28
5.3.11 Edge Cases and Technical considerations . . . 29
5.3.12 Misleding naming, incorrect error types, redundant variables . . . 30

1 About Spearbit

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

Learn more about us at spearbit.com

2 Introduction

Etherex is a decentralized exchange functioning as a metaDEX using the x(3,3) tokenomics model to align incentives between traders, liquidity providers, and token holders.

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

3 Risk classification

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

3.1 Impact

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

3.2 Likelihood

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

3.3 Action required for severity levels

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

4 Executive Summary

Over the course of 49 days in total, Etherex Finance engaged with Spearbit to review the etherex-contracts protocol. In this period of time a total of **36** issues were found.

Summary

Project Name	Etherex Finance
Repository	etherex-contracts
Commit	66c0adc5
Type of Project	DeFi, DEX
Audit Timeline	Aug 26th to Oct 14th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	14	11	3
Gas Optimizations	10	9	1
Informational	12	10	2
Total	**36**	**30**	**6**

5 Findings

5.1 Low Risk

5.1.1 Incorrect fee amount set when gauge is killed

Severity: Low Risk

Context: AccessHub.sol#L257-L258

Description: Expected value is 5%, but `FEE_DENOM = 1_000_000`. Hence, the actual protocol fee set is a lot lower.

Recommendation:

```
  /// @dev set the new fees in the pair to 95/5
-   ramsesV3PoolFactory.setPoolFeeProtocol(pair, 5);
+   ramsesV3PoolFactory.setPoolFeeProtocol(pair, 50_000);
```

Etherex Finance: Fixed in commit 63f3d94a.

Spearbit: Fix verified.

5.1.2 `distribute()` will revert for killed CL gauges

Severity: Low Risk

Context: Voter.sol#L788-L801

Description: There are 2 issues if `distribute()` is called on a killed gauge.

1. The param type for the fee is incorrect, where it should have been `uint24` instead of `uint8`. As a result, the function selector is incorrect, resulting in an EVM revert.

   Proof of Concept:

   ```
   function test_killGaugeAndDistribute() public {
       address gauge = IVoter(VOTER).gaugeForPool(ETH_USDC_POOL);
       address OWNER = ETHEREX_TEAM_MULTISIG;
       vm.startPrank(OWNER);
       address[] memory pairs = new address[](1);
       pairs[0] = ETH_USDC_POOL;
       IAccessHub(ACCESS_HUB).killGauge(pairs);
       // move time forward to next period
       vm.warp(block.timestamp + 1 weeks);
       IVoter(VOTER).distribute(gauge);
   }
   ```

   yields

   ```
   0xAe334f70A7FC44FCC2df9e6A37BC032497Cf80f1::76734e3e(000000000000000000000090e8a5b881d211f418d
   ↪  77ba8978788b62544914b0000000000000000000000000000000000000000000000000000000000000005)
                  [Revert] EvmError: Revert
              [Revert] EvmError: Revert
          [Revert] EvmError: Revert
        [Revert] EvmError: Revert
   ```

2. After changing to `uint24` and etching into `VOTER`, we encounter the 2nd issue on access control where the CLFactory only allows the `AccessHub` to call this method.

   Proof of Concept: Etching the change into `VOTER` and re-running the test:

   ```
   function test_killGaugeAndDistribute() public {
       vm.etch(VOTER, address(new Voter()).code); // changed uint8 to uint24
       address gauge = IVoter(VOTER).gaugeForPool(ETH_USDC_POOL);
       address OWNER = ETHEREX_TEAM_MULTISIG;
   ```

```
        vm.startPrank(OWNER);
        address[] memory pairs = new address[](1);
        pairs[0] = ETH_USDC_POOL;
        IAccessHub(ACCESS_HUB).killGauge(pairs);
        // move time forward to next period
        vm.warp(block.timestamp + 1 weeks);
        IVoter(VOTER).distribute(gauge);
    }
```

yields

```
0xAe334f70A7FC44FCC2df9e6A37BC032497Cf80f1::setPoolFeeProtocol(0x90E8a5b881D211f418d77Ba8978788b
↪  62544914B, 5)
            [Revert] NOT_ACCESSHUB()
          [Revert] NOT_ACCESSHUB()
        [Revert] NOT_ACCESSHUB()
```

Recommendation: The ideal flow is to not change the fee percentage upon killing the gauge, but when the epoch flips. Hence,

1. `setPoolFeeProtocol()` should not be called in `killGauge()`.
2. The appropriate call should be made to the `AccessHub` instead of the `CLFactory`.

Etherex Finance: Fixed in commit 63f3d94a.

Spearbit: Fix verified.

5.1.3 Edge case: Temporary DoS

Severity: Low Risk

Context: VoteModule.sol#L161

Description: It seems that if no-one exited XREX within current period then due to no penalty rewards, rebase of `VOTE MODULE` wont happen (since rebaseThreshold wont reach) at next period start. Thus, if later sometime in running period, penalty is generated and rebase is called then it is possible for unlockTime to roll over to next period.

Proof of Concept:

- `updatePeriodAndRebase` is called on Minter.
- Period gets updated.
- Rebase is called on XREX.
- Lets say rebase threshold is not met so `rebase` does nothing.
- User `A` exits XREX.
- Penalty is charged which is added to `pendingRebase`.
- User `B` calls `rebase` on Minter which eventually calls `rebase` on XREX.
- Lets say threshold is met now so `notifyRewardAmount` is called on Vote Module.
- This updates `rewardSupply` and `unlockTime`.

Now If this rebase was called at the very ending second of current Period then:

- `unlockTime` will rollover to next period which disallows any deposit and withdrawal on next period.
- Once `unlockTime` is over, again if someone exits, create pendingRebase, rebase then unlockTime further increases (Assuming rebase threshold was not met initially again on next period).
- So deposit and withdraw might be blocked for an extra unlockTime.

Impact: Temporary DOS disallowing Deposit and Withdraw for additional `cooldown` duration.

Recommendation:

- Ensure that `unlockTime` should be max upper bound to `currentPeriod` so that it does not rollover to next period.
- Another way would be to set `rebaseThreshold=0` so that `rebase` is completed even with `0` amount.

Etherex Finance: Fixed in commit af738409.

Spearbit: Fix verified.

5.1.4 `updateFeeDistributorForGauge()` doesn't update some mappings

Severity: Low Risk

Context: VoterGovernanceActions.sol#L327-L345

Description: If a fee distributor is updated for a gauge, it doesn't update the `poolForFeeDistributor`, & `feeDistributorForClPair` for CL gauges.

Impact:

- Since `poolForFeeDistributor` is view only, not used internally, thus negligible impact.
- `feeDistributorForClPair` isn't exposed and isn't used internally but is used to fetch fee distributor for existing mapping, thus should be updated.

Recommendation: Update both `poolForFeeDistributor` to the `$.poolForGauge[_gauge]` and `$.feeDistributorForClPair[token0][token1]` to the `_newFeeDistributor` on the `updateFeeDistributorForGauge`.

Note: Fee distributors for a token pair should be same, so if fee distributor is changed for a gauge then it should also be changed for all source killed gauge redirecting to this gauge.

Etherex Finance: Acknowledged.

Spearbit: Acknowledged.

5.1.5 Incorrect state variable used for flash gross fee accounting

Severity: Low Risk

Context: RamsesV3Pool.sol#L728

Description & Recommendation: The variable for `paid1` collected from flash fees should be `grossFeeGrowthGlobal1X128` instead of `grossFeeGrowthGlobal0X128`.

```
- $.grossFeeGrowthGlobal0X128 += FullMath.mulDiv(paid1, FixedPoint128.Q128, _liquidity);
+ $.grossFeeGrowthGlobal1X128 += FullMath.mulDiv(paid1, FixedPoint128.Q128, _liquidity);
```

Etherex Finance: Fixed in commit 5802ee0e.

Spearbit: Fix verified.

5.1.6 It's possible to withdraw from `VoteModule` bypassing `unlockTime`

Severity: Low Risk

Context: VoteModule.sol#L127

Description: Supposed flow is following:

1. Users deposit xRex into `VoteModule` and vote.
2. New epoch starts; Minter calls `xRex.rebase()`. In other words those xRex tokens become a reward to distribute between voters of previous week.

3. `VoteModule` is locked first 12 hours of new period, so that users can't withdraw immediately after "epoch flip". It is safe mechanism to not allow deposit, vote, withdraw, claim reward.

However `VoteModule` lock can be bypassed. Core problem is that it's updated manually during a call to `Minter.updatePeriodAndRebase()`:

```
function updatePeriodAndRebase() external {
    updatePeriod();
    rebase(); // <<
}

function rebase() public {
    /// @dev fetch the data from encoding
    bytes memory data = abi.encodeWithSignature("rebase()");
    /// @dev call the rebase function
    (bool success,) = xRex.call(data); // <<
    require(success, "REBASE_UNSUCCESSFUL");
}

function rebase() external whenNotPaused {
    // ...
    if (
        /// @dev if the rebase is greater than the rebaseThreshold
        period > lastDistributedPeriod && pendingRebase >= rebaseThreshold
    ) {
        // ...
        /// @dev notify the REX rebase
        IVoteModule(VOTE_MODULE).notifyRewardAmount(_temp); // <<
    }
}

function notifyRewardAmount(uint256 amount) external nonReentrant {
    // ...

    /// @dev take the REX from the contract to the voteModule
    underlying.transferFrom(xRex, address(this), amount);

    /// @dev record rewards to the period that just got finalized
    uint256 period = getPeriod();
    rewardSupply[period] += amount;

    /// @dev the timestamp of when people can withdraw next
    /// @dev not DoSable because only xREX can notify
    unlockTime = cooldown + block.timestamp; // <<
}
```

So user can perform following scenario:

1. Deposit xRex to `VoteModule` in last block of period; vote.
2. In next block (which is first block of next period) withdraw xRex from `VoteModule`. `unlockTime` refers to previous periodStart + 12 hour, so it's bypassed.
3. Wait till admin flips period, it sends rewards to `VoteModule`.
4. Finally claim reward in `VoteModule.getPeriodReward()`.

In this case attacker ends up having xRex, which can't be easily converted to Rex because of -50% fee. So it's hard to weaponise and make attack profitable. Still there is following way:

1. Sell xRex via OTC offering discount from claimed profit.
2. xRex is non-transferrable, but it can be bypassed by using deposit and withdraw in Rex33 - this contract is exempted.

3. `REX33.deposit()` is locked in the beginning of new period, so we again need to think how to bypass it.

In REX33 operator firstly claims rewards, swaps it to xRex - and only than unlocks deposits. We've observed unlock delay and here are results (only 4 periods as of writing time):

1. 5.8 hours.
2. 7.6 hours.
3. 0.6 hours.
4. 5.8 hours.

It seems right now operator is manually doing it, but in future it can be automated and hence don't take time.

Proof of Concept:

```
function test_getRewardsViaFastDepositWithdraw() public {
    address RANDOM_USER = makeAddr("random_user");
    uint256 userXRexAmount = 1_000_000e18;
    deal(XREX, RANDOM_USER, userXRexAmount);

    // warp to a minute prior to next period
    uint256 currentTimestamp = vm.getBlockTimestamp();
    uint256 nextPeriod = (currentTimestamp / 1 weeks + 1);
    uint256 nextPeriodTimestamp = nextPeriod * 1 weeks;
    vm.warp(nextPeriodTimestamp - 60);
    vm.startPrank(RANDOM_USER);
    IERC20(XREX).approve(address(VOTE_MODULE), userXRexAmount);
    VoteModule(VOTE_MODULE).depositAll();

    // cast vote
    address[] memory pools = new address[](1);
    pools[0] = ETH_USDC_POOL;
    uint256[] memory weights = new uint256[](1);
    weights[0] = 1;
    IVoter(VOTER).vote(RANDOM_USER, pools, weights);

    // fast forward to next period and withdraw
    skip(60);
    VoteModule(VOTE_MODULE).withdrawAll();
    uint256 xRexBalanceAfterWithdraw = IERC20(XREX).balanceOf(RANDOM_USER);
    // should have received capital back
    assertEq(xRexBalanceAfterWithdraw, userXRexAmount);

    Minter(MINTER).updatePeriodAndRebase();
    // then claim reward, verify that user received rewards
    uint256 rewardAmount = VoteModule(VOTE_MODULE).periodEarned(nextPeriod, RANDOM_USER);
    assertGt(rewardAmount, 0);
    // can claim
    uint256 xRexBalanceBefore = IERC20(XREX).balanceOf(RANDOM_USER);
    VoteModule(VOTE_MODULE).getPeriodReward(nextPeriod);
    uint256 xRexBalanceAfter = IERC20(XREX).balanceOf(RANDOM_USER);
    assertGt(xRexBalanceAfter, xRexBalanceBefore);
    console.log("xREX claimed from reward:", xRexBalanceAfter - xRexBalanceBefore);
    vm.stopPrank();
}
```

Recommendation: VoteModule is immutable, however Voter is upgradeable. Possible solution is to add check into `Voter.poke()` to block withdrawal while period update is not yet executed:

```
function poke(address user) external {
    VoterStorage.VoterState storage $ = VoterStorage.getStorage();
```

```
    // << Block if Minter.UpdatePeriod hasn't been called yet >>
    if (msg.sender == $.voteModule) {
        uint256 currentPeriod = getPeriod();
        uint256 minterActivePeriod = IMinter($.minter).activePeriod();

        if (currentPeriod > minterActivePeriod) {
            revert("NEED_UPDATE_PERIOD");
        }
    }
    // << rest of code >>
}
```

Etherex Finance: Fixed in commit 3e54e464.

Spearbit: Fix verified.

5.1.7 It's impossible to remove vulnerable `nfpManager` without AccessHub upgrade

Severity: Low Risk

Context: AccessHub.sol#L677-L679

Summary: AccessHub have no `removeNfpManager()` functionality, meaning that without AccessHub upgrade it's impossible to remove authorized access to Gauges from active `nfpManager` if it becomes vulnerable.

Finding Description: AccessHub doesn't have its version of `removeNfpManager()`, while `setNfpManager()` doesn't remove the old manager, so it's impossible to remove it without AccessHub upgrade as the corresponding functions are `onlyGovernance`, i.e. `accessHub` limited:

Voter.sol#L49-L51

```
function _onlyGovernance() internal view {
    require(msg.sender == VoterStorage.getStorage().accessHub, Errors.NOT_AUTHORIZED(msg.sender));
}
```

Gauge V3 rewards for the compromised `nfpManager` owned positions are at risk:

GaugeV3.sol#L81-L94

```
/// @dev Ensures the address is an authorized NFP manager
modifier onlyAuthorized(address manager) {
    require(_isValidNfpManager(manager), Errors.NOT_AUTHORIZED_CLAIMER(manager));
    _;
}

/// @dev Allows either voter or authorized NFP managers
modifier onlyAuthorizedOrVoter() {
    require(
        msg.sender == voter || _isValidNfpManager(msg.sender),
        Errors.NOT_AUTHORIZED(msg.sender)
    );
    _;
}
```

GaugeV3.sol#L469-L479

```
function getReward(
    address owner,
    uint256 index,
    int24 tickLower,
    int24 tickUpper,
    address[] memory tokens,
    address receiver
```

```
) external lock onlyAuthorized(owner) {
    require(msg.sender == owner, Errors.NOT_AUTHORIZED(msg.sender));
    _getAllRewards(owner, index, tickLower, tickUpper, tokens, receiver);
}
```

Impact Explanation: In rare occasions when it might be needed to remove a vulnerable `NfpManager` there will be no ability to do so quickly, which can enlarge or even enable the damage caused by it accessing the Gauge rewards for all the users deposited there.

Recommendation: Consider adding `removeNfpManager()` to AccessHub. To enable the change consider removing `syncAllClGauges()` in favor of `syncClGaugesBatch(0, 0)`.

Etherex Finance: Fixed in commit c8022e0f.

Spearbit: Fix verified.

5.1.8 Revived gauges can have stale redirection that will receive emissions instead of them

Severity: Low Risk

Context: VoterGovernanceActions.sol#L121

Summary: When a gauge is being redirected and than revived its `gaugeRedirect` mapping can be stale, i.e. pointing to a no longer active gauge, which will be used in `_distribute()` as `rex` and `xRex` supply destination instead of the revived gauge, blocking its emissions.

Finding Description: The `gaugeRedirect` mapping is set on `redirectEmissions()`, but not updated on `reviveGauge()`, so it becomes stale on revival. Then, this mapping is being unconditionally used in `_distribute()`:

```
address destinationGauge = $.gaugeRedirect[_gauge];
if (destinationGauge == address(0)) {
    destinationGauge = _gauge;
}
```

```
/// @dev check RAM "claimable"
if (_claimable > 0) {
    /// @dev notify emissions
    IGauge(destinationGauge).notifyRewardAmount($.ram, _claimable);
}
/// @dev check xRAM "claimable"
if (_xRamClaimable > 0) {
    /// @dev convert, then notify the xRam
    IXRex(_xRam).convertEmissionsToken(_xRamClaimable);
    IGauge(destinationGauge).notifyRewardAmount(_xRam, _xRamClaimable);
}
```

When the original gauge is not alive the `destinationGauge = $.gaugeRedirect[_gauge]` code is unreachable, while when it is alive, the code resets the gauge to be supplied with emissions to become `$.gaugeRedirect[_gauge]`, which isn't correct as the original gauge is always alive upon reaching this line, so it needn't to be replaced.

Recommendation: Consider resetting the redirection in `reviveGauge()`, e.g.:

```
+ $.gaugeRedirect[_gauge] = address(0);
```

Also, consider removing this `gaugeRedirect` code from `_distribute()` as it happens past `isAlive` check, and is not useful for alive gauges:

```
- address destinationGauge = $.gaugeRedirect[_gauge];
- if (destinationGauge == address(0)) {
-     destinationGauge = _gauge;
- }
```

Etherex Finance: Fixed in commit a5f84ab6.

Spearbit: Fix verified.

5.1.9 Emission update is incorrect

Severity: Low Risk

Context: Minter.sol#L152-L158

Description: Emission changes are allowed in range +-25% of current emission value. The current code implementation is incorrect as shown in POC.

Proof of Concept:

1. Lets say initial emission was `1000`.
2. Since `MAX_DEVIATION` is 25% , thus emission could ideally be changed in between `750 - 1250`.
3. But current code will allow it to be changed even higher say `2000` (100% increase) as shown below:

```
deviation = emissionsMultiplier > _emissionsMultiplier
    ? (emissionsMultiplier - _emissionsMultiplier)
    : (_emissionsMultiplier - emissionsMultiplier);

// deviation = 2000-1000 = 1000

require(deviation <= MAX_DEVIATION, Errors.TOO_HIGH());

// 2000<=2500 which is true

emissionsMultiplier = _emissionsMultiplier;

// emissionsMultiplier = 2000
```

Recommendation: Ensure that emission change is allowed +-25% from current value.

Etherex Finance: Fixed in commit 725dbbe7.

Spearbit: Fix verified.

5.1.10 `xRex` token is not added to `rewards` in `GaugeV3.sol`

Severity: Low Risk

Context: GaugeV3.sol#L124-L135

Description: `xRex` is a default reward token which is meant to be distributed via GaugeV3 along with `Rex`. However `GaugeV3.initialize()` only adds pool tokens to `rewards`:

```
rewards.push(token0); // <<
rewards.push(token1); // <<
(isReward[token0], isReward[token1], isReward[xrex]) = (
    true,
    true,
    true
);
/// @dev if token0 and token1 aren't shadow add shadow in the records
if (token0 != rex && token1 != rex) {
    rewards.push(rex);
    isReward[rex] = true;
}
```

At the same time it enables `xRex` in mapping `isReward`, so in future `xRex` can't be added because check will revert:

```
function addRewards(address reward) external {
    require(msg.sender == voter, Errors.NOT_VOTER(msg.sender));
    if (!isReward[reward]) { // <<
        rewards.push(reward);
        isReward[reward] = true;
        emit RewardAdded(reward);
    }
}
```

As a result, array `rewards` can't contain this reward token. `rewards` is not used in protocol, at this point it's only used offchain.

Recommendation: Add `xRex`:

```
  rewards.push(token0);
  rewards.push(token1);
+ rewards.push(xRex);
  (isReward[token0], isReward[token1], isReward[xrex]) = (
      true,
      true,
      true
  );
```

Etherex Finance: Fixed in commit c86ca7fc.

Spearbit: Fix verified.

5.1.11 Emission rewards are stuck when there is no liquidity in pool

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `RamsesV3Pool.sol` tracks internal accounting, so that `GaugeV3` can reward positions for provided liquidity. During testing it was discovered that emission reward is not distributed when there is no liquidity in pool.

General flow is following:

1. When new period starts, Rex is minted and distributed by Voter between Gauges.
2. Gauge distributes Rex between position owners based on how much and how long in-range liquidity they provided.

Suppose following scenario implemented in test:

1. Period starts. User creates position.
2. After 2 days he burns position.
3. After 1 day creates again.
4. After 1 day burns again.

So in total there were 3 days with in-range positions, and it distributes `3 days / 7 days = 42%` of emission reward. Other 58% are stuck in GaugeV3 and never claimed.

Proof of Concept: Insert this code into folder `test/*`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {RamsesV3Pool} from "../contracts/CL/core/RamsesV3Pool.sol";
import {RamsesV3Factory} from "../contracts/CL/core/RamsesV3Factory.sol";
```

```
import {RamsesV3PoolDeployer} from "../contracts/CL/core/RamsesV3PoolDeployer.sol";
import {NonfungiblePositionManager} from "../contracts/CL/periphery/NonfungiblePositionManager.sol";
import {INonfungiblePositionManager} from
↪  "../contracts/CL/periphery/interfaces/INonfungiblePositionManager.sol";
import {TickMath} from "../contracts/CL/core/libraries/TickMath.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract TestERC20 is ERC20 {
    constructor(uint256 _totalSupply) ERC20("Test Token", "TEST") {
        _mint(msg.sender, _totalSupply);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}

contract FullPeriodPositionTest is Test {
    RamsesV3Factory factory;
    RamsesV3Pool pool;
    NonfungiblePositionManager nfpManager;
    TestERC20 token0;
    TestERC20 token1;

    address user = address(0x1234);
    uint256 constant WEEK = 7 days;

    int24 tickLower = 0;
    int24 tickUpper = 210;


    function setUp() public {
        _setupInfrastructure();
    }

    function testFullPeriodPosition() public {
        vm.warp(10000000);

        uint256 startTime = block.timestamp / WEEK * WEEK;
        uint256 initialPeriod = startTime / WEEK;

        vm.warp(startTime);

        uint256 tokenId1 = _createPosition();

        vm.warp(startTime + 2 days);
        _burnPosition(tokenId1);

        vm.warp(startTime + 3 days);
        uint256 tokenId2 = _createPosition();

        vm.warp(startTime + 4 days);
        _burnPosition(tokenId2);

        vm.warp(startTime + 7 days - 1);

        uint256 result1 = pool.positionPeriodSecondsInRange(
            initialPeriod,
            address(nfpManager),
            tokenId1,
            tickLower,
            tickUpper
```

```
        );

        uint256 result2 = pool.positionPeriodSecondsInRange(
            initialPeriod,
            address(nfpManager),
            tokenId2,
            tickLower,
            tickUpper
        );

        // Calculate emission shares
        uint256 weekX96 = WEEK * (2**96);
        uint256 totalResult = result1 + result2;

        uint256 emissionShare1 = result1 * 10000 / weekX96; // basis points
        uint256 emissionShare2 = result2 * 10000 / weekX96; // basis points
        uint256 totalEmissionShare = totalResult * 10000 / weekX96; // basis points

        console.log("Position 1 emission share:", emissionShare1 / 100, "%");
        console.log("Position 2 emission share:", emissionShare2 / 100, "%");
        console.log("Total emission share:", totalEmissionShare / 100, "%");

        uint256 lostEmission = 10000 - totalEmissionShare;
        console.log("Lost emission:", lostEmission / 100, "%");
    }

    function _setupInfrastructure() internal {
        token0 = new TestERC20(1000000e18);
        token1 = new TestERC20(1000000e18);

        if (address(token0) > address(token1)) {
            (token0, token1) = (token1, token0);
        }

        factory = new RamsesV3Factory(address(0));
        RamsesV3PoolDeployer poolDeployer = new RamsesV3PoolDeployer(address(factory));
        factory.initialize(address(poolDeployer));

        nfpManager = new NonfungiblePositionManager(address(poolDeployer), address(0), address(0),
        ↪  address(0));

        pool = RamsesV3Pool(
            factory.createPool(address(token0), address(token1), 10, TickMath.getSqrtRatioAtTick(200))
        );

        token0.mint(user, 1000000e18);
        token1.mint(user, 1000000e18);

        vm.startPrank(user);
        token0.approve(address(nfpManager), type(uint256).max);
        token1.approve(address(nfpManager), type(uint256).max);
        vm.stopPrank();
    }

    function _createPosition() internal returns (uint256 tokenId) {
        vm.prank(user);
        (tokenId,,,) = nfpManager.mint(
            INonfungiblePositionManager.MintParams({
                token0: address(token0),
                token1: address(token1),
                tickSpacing: 10,
                tickLower: tickLower,
```

```
                tickUpper: tickUpper,
                amount0Desired: 1000e18,
                amount1Desired: 1000e18,
                amount0Min: 0,
                amount1Min: 0,
                recipient: user,
                deadline: block.timestamp + 1000
            })
        );

        return tokenId;
    }

    function _burnPosition(uint256 tokenId) internal {
        (uint128 positionLiquidity,,,,) = pool.positions(
            keccak256(abi.encodePacked(address(nfpManager), tokenId, tickLower, tickUpper))
        );

        vm.prank(user);
        nfpManager.decreaseLiquidity(
            INonfungiblePositionManager.DecreaseLiquidityParams({
                tokenId: tokenId,
                liquidity: positionLiquidity,
                amount0Min: 0,
                amount1Min: 0,
                deadline: block.timestamp + 1000
            })
        );

    }
}
```

Recommendation: There are 2 fixes possible:

1. Without code changes. Always keep your own position in range, so that described scenario never happens.
2. With code changes. Add rescue function to GaugeV3.

Etherex Finance: Acknowledged.

Spearbit: Acknowledged.

5.1.12 `RamsesV3PositionManager`'s liquidity operations can be blocked after period flips

Severity: Low Risk

Context: Position.sol#L224

Summary: `RamsesV3PositionManager`'s liquidity operations can be blocked with reverting `positionPeriodSecondsInRange()` call right after period flips because gauge's active period will be greater than pool's.

Finding Description: Current period is determined based on time only in GaugeV3 contract, while period switch requires `_advancePeriod()` to be run in the pool contract. It is done on pool interactions, but not on position performance reading via `RamsesV3PositionManager._tryClaimRewards()` → `GaugeV3.getRewardForOwner()` → `RamsesV3Pool.positionPeriodSecondsInRange()`, which now happens before calling the pool on liquidity modifications in `RamsesV3PositionManager`.

In the same time `positionPeriodSecondsInRange()` reverts when specified period exceeds current period of the pool, `$.lastPeriod`:

Position.sol#L214-L226

```
/// @notice Get the period seconds in range of a specific position
/// @return periodSecondsInsideX96 seconds the position was in range for the period
function positionPeriodSecondsInRange(PositionPeriodSecondsInRangeParams memory params)
    public
    view
    returns (uint256 periodSecondsInsideX96)
{
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    uint256 currentPeriod = $.lastPeriod;
>>  if (params.period > currentPeriod) revert FTR();

    bytes32 _positionHash = positionHash(params.owner, params.index, params.tickLower,
    ↪  params.tickUpper);
```

This effectively blocks RamsesV3PositionManager's `increaseLiquidity()` and `decreaseLiquidity()` until pool's period is advanced.

Impact Explanation: Temporary unavailability of core RamsesV3PositionManager's liquidity operations. Those can be time dependent and such unavailability can lead to losses for the corresponding users. Can be fixed by running pool's permissionless `_advancePeriod()`, so unavailability is short term, and overall impact is low.

Likelihood Explanation: Can routinely happen on the flip of each period, i.e. weekly. There are no prerequisites, so overall likelihood is medium.

Recommendation: Consider ensuring that pool's period is up to block timestamp before calling `_tryClaimRewards()` for rewards calculation.

Etherex Finance: Fixed in commit 1167bd4a.

Spearbit: Fix verified.

5.1.13 `RamsesV3PositionManager` can excessively slash user

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Previous nfp manager is vulnerable to JIT attacks, therefore now it tracks `positionLastModified` and claims reward during operations. Overall it updates `positionLastModified` during 3 actions:

1. `mint()`.
2. `increaseLiquidity()`.
3. `decreaseLiquidity()`.

And claims reward during 2 actions:

1. `increaseLiquidity()`.
2. `decreaseLiquidity()`.

`positionLastModified` is used to slash user during reward claim. So that now user can't provide liquidity and withdraw it after some blocks, because reward claim will be triggered and hence reward is slashed. In previous version user could wait time before claiming to avoid slash.

```
function validateReward(/*...*/) external view returns (bool) {
    // ...

    // time-based validation (only for the new RamsesV3PositionManager)
    if (_owner == address(nfpManager)) {
        // nft position - use NFPManager's griefing-resistant checkpoint
        uint32 lastModified = nfpManager.positionLastModified(_index); // <<
```

```
        // new positions (never modified) are valid
        if (lastModified == 0) {
            return false; // valid, not slashed
        }

        uint256 elapsedTime = block.timestamp - lastModified;
        return elapsedTime <= timeThreshold; // slash if modified too recently // <<
    }
}
```

However with current design there are 2 situations which overslash user:

1. `increaseLiquidity()` claims rewards. In case user adds liquidity multiple times during short period of time, his reward is forfeited.
2. `positionLastModified` is updated during `decreaseLiquidity()`. In case of multiple calls to `decreaseLiquidity()` it resets timer so user is slashed. Suppose `timeThreshold` is 100; user created position at timestamp 1000; it means at 1100 he will become unslashable. At 1200 he decreases position, so now until 1300 timestamp he is slashable again, which is unfair.

Recommendation: Mention such behaviour explicitly, so that user is aware of it.

Etherex Finance: Acknowledged.

Spearbit: Acknowledged.

5.1.14 CL gauges can have orphaned `feeDistributor`

Severity: Low Risk

Context: VoterGovernanceActions.sol#L339-L341

Summary: One to one gauge to fee distributor correspondence implying logic is applied to both legacy and CL gauges in `updateFeeDistributorForGauge()`, which can yield a live CL gauge with deleted fee distributor whenever there are more than one gauge for some `(token0, token1)` pair and `updateFeeDistributorForGauge()` is run for any of them.

Finding Description: Old fee distributor is deleted on the `updateFeeDistributorForGauge()` call, while it can be still used by some other CL gauges of the same `(token0, token1)` pair:

VoterGovernanceActions.sol#L231-L236

```
/// @dev create a FeeDistributor if needed
address _feeDistributor = $.feeDistributorForClPair[token0][token1];
if (_feeDistributor == address(0)) {
    _feeDistributor =
    ↪  IFeeDistributorFactory($.feeDistributorFactory).createFeeDistributor(_feeCollector);
    $.feeDistributorForClPair[token0][token1] = _feeDistributor;
}
```

Impact Explanation: Functionality that checks for active fee distributor becomes inaccessible, e.g. it will not be possible to remove reward tokens from the pools with this token pair:

VoterGovernanceActions.sol#L322-L328

```
function removeFeeDistributorReward(address _feeDistributor, address reward) external {
    VoterStorage.VoterState storage $ = VoterStorage.getStorage();

    /// @dev ensure the feeDist exists
    require($.feeDistributors.contains(_feeDistributor));
    IFeeDistributor(_feeDistributor).removeReward(reward);
}
```

Likelihood Explanation: Calling `updateFeeDistributorForGauge()` for a gauge is a part of normal workflow, and the only prerequisite is having multiple gauges for the corresponding token pair. In the same time, when only one live gauge is kept for every token pair, the issue requires first running `updateFeeDistributorForGauge()` for current live gauge and then reviving any old gauge from the same pair, which will have orphaned feeDistributor and inaccessible `removeFeeDistributorReward()`. This can be rare, but is probable.

Recommendation: Consider removing old fee distributor for legacy gauges only.

Etherex Finance: Fixed in commit a5613388.

Spearbit: Fix verified.

5.2 Gas Optimization

5.2.1 `lastClaimByToken` can be updated to `currentPeriod` to avoid extra iteration

Severity: Gas Optimization

Context: GaugeV3.sol#L492-L497

Description: `lastClaimByToken` is set to `currentPeriod - 1`, so for the next reward claim, the loop iterates through the previous period again even though all rewards for that period have already been claimed.

Recommendation:

```
- lastClaimByToken[tokens[i]][_positionHash] = currentPeriod - 1;
+ lastClaimByToken[tokens[i]][_positionHash] = currentPeriod;
```

Etherex Finance: Fixed in commit eb7f28b7.

Spearbit: Fix verified.

5.2.2 Cache `r33()` and consolidate sync functions

Severity: Gas Optimization

Context: AccessHub.sol#L693-L694, AccessHub.sol#L704-L705, GaugeV3.sol#L520

Description/Recommendation:

1. `r33()` can be cached assuming it changes infrequently, then add a call in `syncClGaugesBatch()` and `syncAllClGauges()` for syncing.
2. In fact, the separate sync functions can be consolidated into a single sync function, thereby reducing the number of external calls required by the `AccessHub` to sync.

Etherex Finance: Fixed in commit 36ad91a6.

Spearbit: Fix verified.

5.2.3 Redundant status in `WhitelistRevoked` event

Severity: Gas Optimization

Context: VoterGovernanceActions.sol#L73

Description: Only 1 emission of this event and its `status` is always `true`, so it's redundant.

Recommendation:

```
-    emit IVoter.WhitelistRevoked(msg.sender, _token, true);
+    emit IVoter.WhitelistRevoked(msg.sender, _token);

  // in IVoter
-    event WhitelistRevoked(address indexed forbidder, address indexed token, bool status);
+    event WhitelistRevoked(address indexed forbidder, address indexed token);
```

Etherex Finance: Fixed in commit 8fd3b4ac.

Spearbit: Fix verified.

5.2.4 Redundant `augmentGaugeRewardsForPair()` function

Severity: Gas Optimization

Context: AccessHub.sol#L472-L493

Description: Legacy code, no-op.

Recommendation: Remove the referenced lines.

Etherex Finance: Fixed in commit 4ca76515.

Spearbit: Fix verified.

5.2.5 Redundant `ClGaugeFactoryStorage` library

Severity: Gas Optimization

Context: ClGaugeFactoryStorage.sol#L8

Description: Not inherited by `CLGaugeFactory`, likely as a result of refactoring.

Recommendation: Can be removed.

Etherex Finance: Fixed in commit 0e4bedff.

Spearbit: Fix verified.

5.2.6 Unrequired condition

Severity: Gas Optimization

Context: FeeDistributor.sol#L157

Description: Fee Distributor contract never calls `incentivize` function itself thus could be removed from the check.

Recommendation: Remove the `isFeeDistributor` check as shown below:

```
- require(voter.isWhitelisted(token) || voter.isFeeDistributor(msg.sender),
↪  Errors.NOT_WHITELISTED(token));
+ require(voter.isWhitelisted(token) , Errors.NOT_WHITELISTED(token));
```

Etherex Finance: Fixed in commit 31c3f12e.

Spearbit: Fix verified.

5.2.7 `lastClaimByToken` is not updated correctly

Severity: Gas Optimization

Context: FeeDistributor.sol#L273

Description: `_getAllRewards` starts claim from `lastClaim` till `currentPeriod` but `lastClaimByToken[tokens[i]][owner]` is still set to `currentPeriod - 1`. So on next claim it will again claim `currentPeriod - 1` and `currentPeriod`, even though they were already claimed.

Recommendation: Make below changes:

```
- for (uint256 period = lastClaim; period <= currentPeriod; ++period)
+ for (uint256 period = lastClaim+1; period <= currentPeriod; ++period)

  // ...

- lastClaimByToken[tokens[i]][owner] = currentPeriod - 1;
+ lastClaimByToken[tokens[i]][owner] = currentPeriod;
```

Etherex Finance: Acknowledged.

Spearbit: Acknowledged.

5.2.8 `voter` type can be changed to `IVoter`

Severity: Gas Optimization

Context: RamsesV3PositionManager.sol#L78

Description: By changing the `voter` type to `IVoter`, typecasting to it becomes redundant throughout the contract.

Recommendation:

```
- address private voter;
+ IVoter private voter;
```

Etherex Finance: Fixed in commit 4f992828.

Spearbit: Fix verified.

5.2.9 `positionLastModified` can be merged and tightly packed into `Position` struct

Severity: Gas Optimization

Context: RamsesV3PositionManager.sol#L82-L83

Description: Gas costs can be reduced by shrinking the `poolId` from `uint80` to `uint48`, which is still a considerably large number for the number of pool IDs (2.81e14), to accommodate and pack `positionLastModified` into the `Position` struct.

Recommendation:

```
  diff --git a/contracts/CL/periphery/RamsesV3PositionManager.sol
  ↪  b/contracts/CL/periphery/RamsesV3PositionManager.sol
  index ca807a6..a45c93e 100644
-  -- a/contracts/CL/periphery/RamsesV3PositionManager.sol
+  ++ b/contracts/CL/periphery/RamsesV3PositionManager.sol
  @@ -40,7 +40,9 @@ contract RamsesV3PositionManager is
       /// @dev details about the Ramses position
       struct Position {
           /// @dev the ID of the pool with which this token is connected
-          uint80 poolId;
+          uint48 poolId;
+          /// @dev last updated timestamp
+          uint32 lastModified;
           /// @dev the tick range of the position
           int24 tickLower;
           int24 tickUpper;
  @@ -55,10 +57,10 @@ contract RamsesV3PositionManager is
       }

       /// @dev IDs of pools assigned by this contract
-      mapping(address pool => uint80 id) private _poolIds;
+      mapping(address pool => uint48 id) private _poolIds;
```

```
      /// @dev Pool keys by pool ID, to save on SSTOREs for position data
-     mapping(uint80 id => PoolAddress.PoolKey key) private _poolIdToPoolKey;
+     mapping(uint48 id => PoolAddress.PoolKey key) private _poolIdToPoolKey;

      /// @dev The token ID position data
      mapping(uint256 tokenId => Position position) private _positions;
 @@ -66,7 +68,7 @@ contract RamsesV3PositionManager is
      /// @dev The ID of the next token that will be minted. Skips 0
      uint176 private _nextId = 1;
      /// @dev The ID of the next pool that is used for the first time. Skips 0
-     uint80 private _nextPoolId = 1;
+     uint48 private _nextPoolId = 1;

      /// @dev The address of the token descriptor contract, which handles generating token URIs for
      ↪  position tokens
      address private immutable _tokenDescriptor;
 @@ -75,13 +77,10 @@ contract RamsesV3PositionManager is
      IAccessHub private immutable accessHub;

      /// @dev the address of the voter contract
-     address private voter;
+     IVoter private voter;
      address private ram;
      address private xRam;

-     /// @dev cache the last modified timestamp for each tokenId
-     mapping(uint256 => uint32) public positionLastModified;
-
      constructor(
          address _deployer,
          address _WETH9,
 @@ -130,7 +129,7 @@ contract RamsesV3PositionManager is
      }

      /// @dev Caches a pool key
-     function cachePoolKey(address pool, PoolAddress.PoolKey memory poolKey) private returns (uint80
↪  poolId) {
+     function cachePoolKey(address pool, PoolAddress.PoolKey memory poolKey) private returns (uint48
↪  poolId) {
          poolId = _poolIds[pool];
          if (poolId == 0) {
              _poolIds[pool] = (poolId = _nextPoolId++);
 @@ -174,13 +173,14 @@ contract RamsesV3PositionManager is
          (, uint256 feeGrowthInside0LastX128, uint256 feeGrowthInside1LastX128, , ) =
          ↪  pool.positions(positionKey);

          /// @dev idempotent set
-         uint80 poolId = cachePoolKey(
+         uint48 poolId = cachePoolKey(
              address(pool),
              PoolAddress.PoolKey({token0: params.token0, token1: params.token1, tickSpacing:
              ↪  params.tickSpacing})
          );

          _positions[tokenId] = Position({
              poolId: poolId,
+             lastModified: uint32(block.timestamp),
              tickLower: params.tickLower,
              tickUpper: params.tickUpper,
              liquidity: liquidity,
 @@ -190,8 +190,6 @@ contract RamsesV3PositionManager is
```

```
            tokensOwed1: 0
        });

-       positionLastModified[tokenId] = uint32(block.timestamp);
-
        emit IncreaseLiquidity(tokenId, liquidity, amount0, amount1);
    }

@@ -207,8 +205,8 @@ contract RamsesV3PositionManager is
    }

    function _tryClaimRewards(uint256 tokenId, IRamsesV3Pool pool) private {
-       if (voter != address(0)) {
-           address gauge = IVoter(voter).gaugeForPool(address(pool));
+       if (voter != IVoter(address(0))) {
+           address gauge = voter.gaugeForPool(address(pool));
            if (gauge != address(0)) {
                // only claim protocol tokens to prevent gas bomb attacks
                address[] memory rewardTokens = new address[](2);
@@ -290,7 +288,7 @@ contract RamsesV3PositionManager is
        }

        // checkpoint
-       positionLastModified[params.tokenId] = uint32(block.timestamp);
+       position.lastModified = uint32(block.timestamp);

        emit IncreaseLiquidity(params.tokenId, liquidity, amount0, amount1);
    }
@@ -360,7 +358,7 @@ contract RamsesV3PositionManager is
        }

        // checkpoint
-       positionLastModified[params.tokenId] = uint32(block.timestamp);
+       position.lastModified = uint32(block.timestamp);

        emit DecreaseLiquidity(params.tokenId, params.liquidity, amount0, amount1);
    }
@@ -441,7 +439,7 @@ contract RamsesV3PositionManager is


        PoolAddress.PoolKey memory poolKey = _poolIdToPoolKey[position.poolId];
-       IGaugeV3 gauge = IGaugeV3(IVoter(voter).gaugeForPool(PoolAddress.computeAddress(deployer,
↪  poolKey)));
+       IGaugeV3 gauge = IGaugeV3(voter.gaugeForPool(PoolAddress.computeAddress(deployer, poolKey)));

        gauge.getRewardForOwner(tokenId, tokens);
    }
@@ -456,11 +454,16 @@ contract RamsesV3PositionManager is
        Position storage position = _positions[tokenId];

        PoolAddress.PoolKey memory poolKey = _poolIdToPoolKey[position.poolId];
-       address gauge = IVoter(voter).gaugeForPool(PoolAddress.computeAddress(deployer, poolKey));
+       address gauge = voter.gaugeForPool(PoolAddress.computeAddress(deployer, poolKey));

        IGaugeV3(gauge).getPeriodReward(period, tokens, address(this), tokenId, position.tickLower,
        ↪  position.tickUpper, receiver);
    }

+   /// @inheritdoc IRamsesV3PositionManager
+   function positionLastModified(uint256 tokenId) external view returns (uint32) {
+       return _positions[tokenId].lastModified;
+   }
```

```
+
       /// @inheritdoc INonfungiblePositionManager
       function burn(uint256 tokenId) external payable override isAuthorizedForToken(tokenId) {
           Position storage position = _positions[tokenId];
  @@ -471,13 +474,13 @@ contract RamsesV3PositionManager is

       /// @notice extra function that allows for the 2-step deployment of CL first, then governance
       ↪  later
       /// @dev gated to the timelock
-      function setVoter(address _voter) external {
+      function setVoter(IVoter _voter) external {
           require(msg.sender == accessHub.timelock());
           voter = _voter;

           // cache rex and xrex addresses to save gas on every claim
-          ram = IVoter(_voter).ram();
-          xRam = IVoter(_voter).xRam();
+          ram = _voter.ram();
+          xRam = _voter.xRam();
       }

       //** Overrides */
```

Notably, minting positions decreased by 21k gas.

```
- "decreasing liquidity": "606177",
- "increasing liquidity": "848860",
- "minting position": "599980"
+ "decreasing liquidity": "605719",
+ "increasing liquidity": "848668",
+ "minting position": "578284"
```

Etherex Finance: Fixed in commit 4f992828.

Spearbit: Fix verified.

5.2.10 Position key computation can be performed in `assembly`

Severity: Gas Optimization

Context: PositionKey.sol#L5-L9

Description: By slightly altering UniV4's `Position` library's implementation, ~137 gas can be saved for position mints and modifications. The structure is very similar, where they append a `salt` at the end, while the `index` here is between the `owner` and tick fields.

Recommendation:

```
library PositionKey {
    /// @dev Returns the key of the position in the core library
    function compute(address owner, uint256 index, int24 tickLower, int24 tickUpper)
        internal
        pure
        returns (bytes32 positionKey)
    {
        // positionKey = keccak256(abi.encodePacked(owner, index, tickLower, tickUpper)
        assembly ("memory-safe") {
            let fmp := mload(0x40)
            mstore(add(fmp, 0x26), tickUpper) // [0x43, 0x46)
            mstore(add(fmp, 0x23), tickLower) // [0x40, 0x43))
            mstore(add(fmp, 0x20), index) // [0x20, 0x40)
            mstore(fmp, owner) // [0x0c, 0x20)
```

```
            positionKey := keccak256(add(fmp, 0x0c), 0x3a) // len is 58 bytes

            // now clean the memory we used
            mstore(add(fmp, 0x40), 0) // fmp+0x40 held tickLower, tickUpper
            mstore(add(fmp, 0x20), 0) // fmp+0x20 held index
            mstore(fmp, 0) // fmp held owner
        }
    }
}
```

Etherex Finance: Fixed in commit 4dc330c7.

Spearbit: Fix verified.

5.3 Informational

5.3.1 Incorrect comment

Severity: Informational

Context: IRamsesV3PoolState.sol#L138, Position.sol#L215

Description/Recommendation:

```
- /// @return periodSecondsInsideX96 seconds the position was not in range for the period
+ /// @return periodSecondsInsideX96 seconds the position was in range for the period
```

Etherex Finance: Fixed in commit 3f844674.

Spearbit: Fix verified.

5.3.2 Incorrect errors

Severity: Informational

Context: GaugeV3.sol#L233

Description/Recommendation:

- VoterGovernanceActions.sol#L277-L278: `NO_GAUGE(pool)` doesn't really fit here, as there's the possibility of inactive gauges. Maybe something like `GAUGE_INACTIVE`, meaning that it's not active for this token0/token1 pair:
- GaugeV3.sol#L233: Revert is for attempt to reward past periods, so `CANT_CLAIM_FUTURE` doesn't make much sense. Perhaps something like `CANT_REWARD_PAST` would be more appropriate.

Etherex Finance: Fixed in commits 73c826af and eab33960.

Spearbit: Fix verified.

5.3.3 Incorrect fee amount logged

Severity: Informational

Context: FeeCollector.sol#L63-L64, FeeCollector.sol#L80-L83

Description: `pushable0` & `pushable1` should be decremented because 1 wei is kept in the pool, so the fees collected tracked will be off by 1 wei.

Recommendation: Use the return values from `collectProtocol()`.

Etherex Finance: Fixed in commit d69a630d.

Spearbit: Fix verified.

5.3.4 `Abstained` event isn't meaningful

Severity: Informational

Context: Voter.sol#L325

Description: The `Abstained` event emits the `owner` as `address(0)` instead of the owner. Furthermore, it's emitted per iteration without context of the pool, so there isn't sufficient context.

Recommendation:

1. The owner emitted be `user`.
2. The pool should also be emitted in the event, otherwise it should be emitted outside the loop.

Etherex Finance: Fixed in commit 082cbdd0. Removed the `Abstained` event entirely.

Spearbit: Fix verified.

5.3.5 `GaugeV3.getPeriodReward()` is not called by position manager

Severity: Informational

Context: GaugeV3.sol#L392-L400

Description: Function `getPeriodReward()` allows to claim reward for any past period, that is designed to be used in case for some reason user is not able to claim via usual way, for example due to out-of-gas error during loops.

Problem is that it's callable only by position manager, but `RamsesV3PositionManager` does not call it:

```
function getPeriodReward(
    uint256 period,
    address[] calldata tokens,
    address owner,
    uint256 index,
    int24 tickLower,
    int24 tickUpper,
    address receiver
) external override lock onlyAuthorized(owner) {
    require(msg.sender == owner, Errors.NOT_AUTHORIZED(msg.sender)); // <<

    bytes32 _positionHash = positionHash(owner, index, tickLower, tickUpper);

    for (uint256 i = 0; i < tokens.length; ++i) {
        if (period < _blockTimestamp() / WEEK) {
            lastClaimByToken[tokens[i]][_positionHash] = period;
        }

        _getReward(period, tokens[i], owner, index, tickLower, tickUpper, _positionHash, receiver);
    }
}
```

Recommendation: Implement missing function in `RamsesV3PositionManager`.

Etherex Finance: Fixed in commit fcb680c3.

Spearbit: Fix verified.

5.3.6 `GaugeV3.getPeriodReward()` should not update `lastClaimByToken`

Severity: Informational

Context: GaugeV3.sol#L407

Description: GaugeV3 is tracking `lastClaimByToken` variable to be able to claim only pending periods. For example: `lastClaimByToken = 5`, current active period is 9; it means that usual claim will loop over periods [5, 9].

```
function _getAllRewards(
    address owner,
    uint256 index,
    int24 tickLower,
    int24 tickUpper,
    address[] memory tokens,
    address receiver
) internal {
    bytes32 _positionHash = positionHash(owner, index, tickLower, tickUpper);
    uint256 currentPeriod = _blockTimestamp() / WEEK;
    uint256 lastClaim;

    for (uint256 i = 0; i < tokens.length; ++i) {
        lastClaim = Math.max(lastClaimByToken[tokens[i]][_positionHash], firstPeriod);
        for (uint256 period = lastClaim; period <= currentPeriod; ++period) { // <<
            _getReward(period, tokens[i], owner, index, tickLower, tickUpper, _positionHash, receiver);
        }
        lastClaimByToken[tokens[i]][_positionHash] = currentPeriod - 1;
    }
}
```

Function `GaugeV3.getPeriodReward()` claims specific past period and also updates `lastClaimByToken`:

```
function getPeriodReward(
    uint256 period,
    address[] calldata tokens,
    address owner,
    uint256 index,
    int24 tickLower,
    int24 tickUpper,
    address receiver
) external override lock onlyAuthorized(owner) {
    require(msg.sender == owner, Errors.NOT_AUTHORIZED(msg.sender));

    bytes32 _positionHash = positionHash(owner, index, tickLower, tickUpper);

    for (uint256 i = 0; i < tokens.length; ++i) {
        if (period < _blockTimestamp() / WEEK) {
            lastClaimByToken[tokens[i]][_positionHash] = period; // <<
        }

        _getReward(period, tokens[i], owner, index, tickLower, tickUpper, _positionHash, receiver);
    }
}
```

So suppose following scenario:

1. `lastClaimByToken = 5`, current active period is 9.
2. `GaugeV3.getPeriodReward()` is called with period 7. Now `lastClaimByToken = 7`.
3. Usual claim loop will process periods 7, 8, 9. And it misses 6 because of unnecessary update of `lastClaimByToken` before.

Recommendation: Consider not updating `lastClaimByToken`.

Etherex Finance: Fixed in commit 39107ca8.

Spearbit: Fix verified.

5.3.7 Function `Voter.vote()` can be simplified

Severity: Informational

Context: Voter.sol#L397-L402

Description: There is legacy code where you convert `calldata` array into `memory`:

```
function vote(address user, address[] calldata _pools, uint256[] calldata _weights) external {
    VoterStorage.VoterState storage $ = VoterStorage.getStorage();

    /// @dev ensure that the arrays length matches and that the length is > 0
    require(_pools.length > 0 && _pools.length == _weights.length, Errors.LENGTH_MISMATCH());
    /// @dev if the caller isn't the user...
    if (msg.sender != user) {
        /// @dev ...require they are authorized to be a delegate
        require(
            IVoteModule($.voteModule).isDelegateFor(msg.sender, user) || msg.sender == $.accessHub,
            Errors.NOT_AUTHORIZED(msg.sender)
        );
    }
    /// @dev make a memory array of votedPools
    address[] memory votedPools = new address[](_pools.length); // <<
    /// @dev loop through and populate the array
    for (uint256 i = 0; i < _pools.length; ++i) { // <<
        votedPools[i] = _pools[i];
    }

    /// @dev cast new votes
    _vote(user, votedPools, _weights);
}
```

There is no need to do it, because you can submit `_pools` directly into `_vote()`.

Recommendation: Do not copy `calldata _pools`, pass it directly to `_vote()`.

Etherex Finance: Fixed in commit 41baa02e.

Spearbit: Fix verified.

5.3.8 `feeRecipient` should be directly fetched from `pool`

Severity: Informational

Context: Voter.sol#L825

Description: If `AccessHub.setFeeRecipientLegacyBatched()` is called to update the fee recipients, it will not update the `feeRecipientForPair` mapping. Hence, it would be more appropriate to call `IPair(pool).feeRecipient()`.

Recommendation:

```
- IFeeRecipient(IFeeRecipientFactory($.feeRecipientFactory).feeRecipientForPair(pool)).notifyFees();
+ IFeeRecipient(IPair(pool).feeRecipient()).notifyFees();
```

Etherex Finance: Fixed in commit 2b3b2f72.

Spearbit: Fix verified.

5.3.9 Improper Access Control in `FeeCollector` and `AccessHub` interaction

Severity: Informational

Context: FeeCollector.sol#L30-L48

Description: `AccessHub` contract manages configuration in protocol. Let's take a look that it also manages `FeeCollector`:

```
/// @inheritdoc IAccessHub
function setTreasuryInFeeCollector(address newTreasury) external onlyRole(PROTOCOL_OPERATOR) {
    feeCollector.setTreasury(newTreasury);
}

/// @inheritdoc IAccessHub
function setTreasuryFeesInFeeCollector(uint256 _treasuryFees) external onlyRole(PROTOCOL_OPERATOR) {
    feeCollector.setTreasuryFees(_treasuryFees);
}
```

However `FeeCollector` is actually managed by Treasury:

```
/// @dev Prevents calling a function from anyone except the treasury
modifier onlyTreasury() {
    require(msg.sender == treasury, Errors.NOT_AUTHORIZED(msg.sender));
    _;
}

/// @inheritdoc IFeeCollector
function setTreasury(address _treasury) external override onlyTreasury {
    emit TreasuryChanged(treasury, _treasury);

    treasury = _treasury;
}

/// @inheritdoc IFeeCollector
function setTreasuryFees(uint256 _treasuryFees) external override onlyTreasury {
    require(_treasuryFees <= BASIS, Errors.FEE_TOO_LARGE());
    emit TreasuryFeesChanged(treasuryFees, _treasuryFees);

    treasuryFees = _treasuryFees;
}
```

Usually Treasury in protocol refers to protocol's Safe multisig, however in this case Treasury is contract `RamsesTreasuryHelper` which doesn't have functionality to configure `FeeCollector`. This can be observed in currently live `FeeCollector`, which points to `RamsesTreasuryHelper`

As a result, there is no way to update `treasuryFees` variable. This variable means how much of swap fee in `RamsesV3Pool` belongs to protocol, all other fee is distributed between users for voting. In other words, protocol can't enable protocol fee (current value is 0).

Recommendation: In GitHub update `FeeCollector` to use `onlyAccessHub` modifier. However contracts are live and `RamsesTreasuryHelper` is upgradeable. You can add and call `setTreasury()` to set `treasury` to `AccessHub`.

Etherex Finance: Fixed in commit b47657ed.

Spearbit: Fix verified.

5.3.10 Unused Variable

Severity: Informational

Context: Gauge.sol#L36

Description/Recommendation: `xRex` variable in `Gauge.sol` is not used and can be removed.

Etherex Finance: Fixed in commit 33496ad1.

Spearbit: Fix verified.

5.3.11 Edge Cases and Technical considerations

Severity: Informational

Context: AccessHub.sol#L269-L272, AccessHub.sol#L515-L517, ClGaugeFactory.sol#L63, GaugeV3.sol#L196, RamsesV3PositionManager.sol#L193, RamsesV3PositionManager.sol#L471-L477, Etherex.sol#L38-L41, VoterGovernanceActions.sol#L105-L108, VoterGovernanceActions.sol#L115, VoterGovernanceActions.sol#L269, Voter.sol#L102, Voter.sol#L447, Voter.sol#L643, Voter.sol#L762, REX33.sol#L218-L220, REX33.sol#L264, XRex.sol#L260

Description:

- REX33.sol: XRex token is normally non transferrable but a user can simply deposit xRex to rex33 and then withdraw to destination receiver.
- REX33.sol: Deposit via REX33 is only allowed post unlock which is triggered in 2 conditions:
  1. `periodUnlockStatus[getPeriod()]` is true.
  2. `timeLeftInPeriod > 1 hours`.

  A User who deposits directly to Vote Module and don't use REX33 can do so when period is unlocked and is not constrained by 2nd condition (`timeLeftInPeriod > 1 hours`).
- RamsesV3PositionManager.sol: When we transfer NFT, `_update` does not call `_tryClaimRewards` which means sender rewards till date also get transferred to recipient.
- Etherex.sol: If `Minter.sol` ever need to be changed, it would be an issue since current `Minter.sol` has no way to transfer minting rights to new `Minter.sol`.
- sVoterGovernanceActions.sol: If no one call distribute on gauge for multiple period and then it gets killed, all rewards goes to governor.
- VoterGovernanceActions.sol: Consider below scenario:
  - Lets say epoch `E1` has just started.
  - `Revive` is called to revive one of the killed gauge `G1`.
  - This sets `95`% fees to LP, `0`% emission.
  - `distribute` is called on gauge `G1`.
  - This resets the percentage back to `0`% fees to LP, `100`% emission which is unexpected.

  So basically first call `distribute` so that gauge `G1` is initially ignored (since its killed) then `revive` it so that fees is set to 95% fees to LP, 0% emission for the running epoch.

  Once next epoch starts, distribute sets it back to 0% fees to LP, 100% emission.
- AccessHub.sol: If gauge gets killed and revived in same period then fee protocol would manually need to be reset back to 100% for current period.
- AccessHub.sol: Operator should ensure that `setTreasuryFeesInFeeCollector` is only called post `collectProtocolFees` has been called for all pools. Else existing protocol fees would use this new treasury fees.
- VoterGovernanceActions.sol: `redirectEmissions` could unintentionally allow stealing treasury fees.
  - Gauge G1 was killed.
  - Pool linked with `G1` has more swaps making more protocol fees which is unclaimed.
  - Gauge `G2` is created.
  - `redirectEmissions` is called which makes `$.gaugeRedirect[G1] = G2;`.
  - Now if `collectProtocolFees` is called then fees wont go to treasury.
  - But if `collectProtocolFees` was called before `redirectEmissions` then fees would have gone to treasury.

Note: The loss to treasury will reduce considerably if `redirectEmissions` was called on next period as `distribute` on old gauge would collect fees till date and change protocol fees to only 5%.

- Voter.sol: Ensure that all killed and revived gauges are immediately called on epoch flip else gauges may work with incorrect protocol fee on next epoch.
  1. Gauge `G1` was killed on epoch `1`.
  2. Protocol fee remains `100%`.
  3. If call to `distribute` is made on epoch `3` then all `100%` fee collected on epoch `2` would be treated as protocol fee and would go to treasury causing loss to users.
  4. Same applies for reviving gauges.
- GaugeV3.sol: If pool and gauge was created in mid period and User added incentive by adding rewards via `notifyRewardAmount` then rewards from period start till first active mint will get stuck.
- RamsesV3PositionManager.sol:
  - Genuine frequent mint/burn operation could cause loss of rewards via slashing.
  - Suppose a whale want to mint liquidity and after a while wants to add more. In this case, its better for whale to use wait for threshold time or use another index since else `positionLastModified` may not cross threshold and whale rewards would get slashed.
  - Similar case for burn.
- Voter.sol: Suppose user votes for gauges, then one of them is killed, then he calls `poke()`. In this case it won't use full voting power. So actually user should manually revote with new weights or call `poke()` twice.
- There are multiple config functions that can be called only by `AccessHub`, however `AccessHub` never calls them:
  - Voter.transferOwnership().
  - ClGaugeFactory.setVoter().
  - XRex.setRebaseThreshold().
  - Voter.removeNfpManager().

Etherex Finance: Acknowledged.

Spearbit: Acknowledged.

5.3.12 Misleding naming, incorrect error types, redundant variables

Severity: Informational

Context: VoterGovernanceActions.sol#L277-L278

Description:

- Naming:
  1. `validateReward` returns `true` when slashing criteria has been met. This isn't aligned with the name of the function, as `true` means that reward wasn't validated:

     RewardValidator.sol#L156-L157

     ```
     /// @return true if the position should be slashed (blacklisted or too recently modified)
     function validateReward(
     ```

     Also, `address _pool` is missed in `@param`.
- Errors:

1. `NO_GAUGE(pool)` error is not correct for `require(_gaugesForClPair.contains(destinationGauge)` check in `redirectEmissions()`.

```
/// @dev require the destination gauge to be of the same token0/token1 pair
require(_gaugesForClPair.contains(destinationGauge), Errors.NO_GAUGE(destinationGauge));
```

```
/// @notice Thrown when pool doesn't have an associated gauge
/// @param pool The address of the pool
error NO_GAUGE(address pool);
```

2. `NOT_WHITELISTED(from)` error isn't correct for xRex's `_update()`.

   Errors.sol#L28-L30

```
/// @notice Thrown when token is not whitelisted
/// @param token The address of the non-whitelisted token
error NOT_WHITELISTED(address token);
```

- Cleaning up:

1. Using `PRECISION` simultaneously on both sides doesn't change the rounding in VoteModule's `periodEarned()`, but somewhat adds to the probability of overflows.

```
amount = ((votingPowerUsed * PRECISION * periodRewardSupply) / (totalVotesPerPeriod *
↪  PRECISION))
```

   Also, `PRECISION` is declared, but isn't used in GaugeV3.

Recommendation:

1. Consider either renaming, e.g. making it `isRewardSlashable()`, or reversing the logic, updating `@param`.
2. Consider creating a new error or use something closer, e.g. `GAUGE_INACTIVE`, meaning that it's not active for this token0/token1 pair:

```
/// @notice Thrown when attempting to interact with an inactive gauge
/// @param gauge The address of the gauge
error GAUGE_INACTIVE(address gauge);
```

3. Consider using `NOT_AUTHORIZED(address caller)` or a new error.
4. Consider removing `PRECISION` in both cases.

Etherex Finance: Acknowledged.

Spearbit: Acknowledged.