CANTINA

Incentive contracts

Competition

September 16, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

ZeroLend is one of the fastest-growing multi-chain lending protocols, which focuses on Liquid Restaking Tokens (LRTs), Real World Assets (RWAs), privacy, and account abstraction.

From Jan 8th to Jan 25th Cantina hosted a competition based on incentive-contracts. The participants identified a total of **455** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 6
- Medium Risk: 17
- Low Risk: 193
- Gas Optimizations: 48
- Informational: 191

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 It is impossible to claim rewards in `zlrewardscontroller.sol`

Submitted by osmanozdemir1, also found by Federico Bianucci, Sujith Somraaj, jovi.eth, 0xhashiman, tnch, bin2chen, elhaj, Chinmay Farkya, pauleth, Naveen Kumar J - 1nc0gn170, wafflemakr, 0xarno and shaka

Severity: High Risk

Context: ZLRewardsController.sol#L484

Description: `ZLRewarsController` contract is responsible for storing reward bearing tokens' data, users' data and allocation percentages for different tokens. All reward bearing tokens earn some rewards based on accumulated rewards per token.

Users get proportional amount of rewards by holding these reward bearing tokens, and they can claim their reward with the ZLRewardsController.claim() function.

However, this function will always revert. Let's examine said function:

```
function claim(
    address _user,
    address[] memory _tokens
) public whenNotPaused {

    // SKIPPED FOR BREVITY

    _vestTokens(_user, pending);
}
```

The `claim` function performs some checks and actions (*calculates reward debt and user pending reward etc*), and then calls the _vestTokens function.

```
function _vestTokens(address _user, uint256 _amount) internal {
    if (_amount == 0) revert NothingToVest();
    streamedVesting.createVestFor(_user, _amount); //@audit-issue streamedVesting contract will try to burn
↪   vestedTokens from this contract. This contract never approved the StreamedVesting contract
}
```

The `_vestTokens` function then will call the `streamedVesting.createVestFor` function, which uses the `msg.sender` as the `from` address, and the msg.sender is the `ZLRewardsController` contract.

```
function createVestFor(address to, uint256 amount) external whenNotPaused {
    _createVest(msg.sender, to, amount); //@audit msg.sender is ZLRewardsController contract
}
```

```
function _createVest(
    address from,
    address to,
    uint256 amount
) internal whenNotPaused {
    vestedToken.burnFrom(from, amount); //@audit-issue "from" is ZLRewardsController contract. That contract
↪   is never approved StreamedVesting as "spender"
    lastId++;

    // SKIPPED FOR BREVITY
}
```

As we can see above, the first step during the `_createVest` function is burning `vestedTokens` from the "from" address (which is `ZLRewardsController`) with this line: `vestedToken.burnFrom(from, amount);`

`vestedToken` is an `ERC20Burnable`, and let's check the burnFrom function in the OpenZeppelin implementation:

```
function burnFrom(address account, uint256 value) public virtual {
    _spendAllowance(account, _msgSender(), value);
    _burn(account, value);
}
```

As we can see here, it requires allowance. `ZLRewardsControooler.claim()` will always revert with ERC20InsufficientAllowance error.

`ZLRewardsController` contract MUST approve the `StreamedVesting` contract as spender for this claim functionality to work.

Similar approvals are made in this protocol at StreamedVesting.sol#L56 (`streamedVesting` approves the `ZeroLocker` contract as spender) and at BonusPool.sol#L24 (`bonusPool` approves the `sreamedVesting` contract as spender) during initialization.

Likelihood is high as it will always happen. Impact is high since this is one of the core functions and the reward mechanism of the protocol is totally broken, which makes it a critical issue.

Proof of concept:

Before running the test we need to do a slight change in the protocol's fixture. We are going to change the pool configurator to make testing easier. Perform the change below in the `test/fixtures/core.ts` test file:

```
  await zLRewardsController.initialize(
-     lending.configurator.target, // address _poolConfigurator,
+     owner.address, // address _poolConfigurator,
      vesting.target, // IStreamedVesting _streamedVesting,
      locker.target, // IZeroLocker _locker,
      1000, // uint256 _rewardsPerSecond,
      token.target, // address _rdntToken,
      0 // uint256 _endingTimeCadence
  );
```

After changing these lines, do the following steps:

- Create an empty file in the `test` folder and name it `something.ts`.
- Copy and paste the snippet below into the newly created file.
- Run it with `npx hardhat test --grep "Always reverts when claiming"`.

```
import { expect } from "chai";
import { ethers } from "hardhat";
import {
  loadFixture,
  time,
} from "@nomicfoundation/hardhat-toolbox/network-helpers";

import { e18, deployCore as fixture } from "./fixtures/core";

describe("RewardsController", () => {
    it("Always reverts when claiming", async function () {
        const {
          token: rewardToken,
          vestedToken,
          owner,
          zLRewardsController,
          otherAccount: user,
        } = await loadFixture(fixture);

        // First, transfer some vestedToken token to zlRewardsController address.
        // StreamedVesting contract will try to burn them from "zlRewardsController" during claim.
        await vestedToken.transfer(zLRewardsController, e18 * 100000000n);

        // Configurator adds pool to the contract (configurator is owner).
        // Note: I used regular token as rewardToken here instead of deploying a new mock token to make
↪  testing easier.
        await zLRewardsController.connect(owner).addPool(rewardToken, e18 * 100000n);
        expect(await zLRewardsController.poolLength()).eq(1);

        // Owner registers reward deposit and starts.
        await zLRewardsController.connect(owner).registerRewardDeposit(e18 * 100000000n);
        await zLRewardsController.connect(owner).start();

        // Transfer some reward tokens to user and register the user. "afterLockUpdate" function will register
↪  the user
        await rewardToken.transfer(user.address, e18 * 100n);
        await zLRewardsController.afterLockUpdate(user.address);
```

```
        // fast forward time
        await time.increase(86400 * 30 * 3);

        // Check user's pending reward
        const pendingReward = await zLRewardsController.allPendingRewards(user.address);
        expect(pendingReward).greaterThan(0);
        console.log("pend: ", pendingReward);

        // ----------------------ACTION----------------
        // User tries to claim.
        // It will revert with insufficient allowance error
        expect(zLRewardsController.connect(user).claimAll(user.address)).revertedWith("ERC20: insufficient
↪  allowance");
    });
});
```

Recommendation: Approve the `streamedVesting` contract as spender during initialize.

```
+ // Inside the initialize
+ vestedToken.approve(address(steamedVesting), type(uint256).max)
```

3.1.2 `zlrewardscontroller` contract is incompatible with zerolend's own reward bearing tokens

Submitted by osmanozdemir1, also found by shaka

Severity: High Risk

Context: ZLRewardsController.sol#L510

Description: The `ZLRewardsController` contract is responsible for managing rewards. Owner can configure this contract by adding reward bearing tokens, deciding allocation points for each token and depositing rewards. Users can earn rewards by holding these reward bearing tokens and claim their rewards.

The main logic of reward tracking is based on `MasterChef` contract. User's information of how much reward per token is accumulated and the reward debt is tracked here. Based on this logic, every action that changes user balance (e.g. transfer, mint or burn) should invoke a hook, and the user info should be updated. This way users will always get fair share of total rewards based on their balances. This hook in `ZLRewardsController` contract is _handleActionAfterForToken function, which is called in `handleActionAfter` function.

```
function handleActionAfter( //@audit Reward bearing tokens should call this function.
    address _user,
    uint256 _balance,
    uint256 _totalSupply
) external {
    if (!validRTokens[msg.sender] && msg.sender != address(streamedVesting))
        revert NotRTokenOrMfd();

    if (_user == address(streamedVesting)) return;

    _handleActionAfterForToken(msg.sender, _user, _balance, _totalSupply);
}
```

This was introduction part about the logic. Now, let's check why this contract is not incompatible with the protocol itself.

As we can see at ZLRewardsController.sol#L17, `ZLRewardsController` contract is based on Radiant Capital's ChefIncentivesController.sol contract. You can see the `handleActionAfter` function in Radiant Capital's repo at ChefIncentivesController.sol#L611-L632.

Both Radiant Capital and ZeroLend's core protocol contracts are forks from Aave. However, it seems like Radiant Capital has changed their token implementations. They changed the regular `handleAction` hook from Aave to `handleActionAfter`.

Check out Radiant Capital's token implementation.

Radiant's token implementation _transfer function looks like this:

```
// skipped for brevity
if (address(_getIncentivesController()) != address(0)) {
    uint256 currentTotalSupply = _totalSupply;
    _getIncentivesController().handleActionAfter(sender, _balances[sender], currentTotalSupply);
    if (sender != recipient) {
            _getIncentivesController().handleActionAfter(recipient, _balances[recipient], currentTotalSupply);
    }
}
```

Their reward bearing tokens call their incentives controller contract's `handleActionAfter` hook after every balance changing action, which is exactly how it should be.

Now, for Zerolend, let's check the core contract's token implementation.

Here you can also see the deployed versions in Zksync Era.

ZeroLend's token implementation is also forked from Aave but unlike Radiant Capital, the hook is unchanged. The _transfer function of ZeroLend's token implementation calls `handleAction`:

```
function _transfer(address sender, address recipient, uint128 amount) internal virtual {
    // skipped for brevity

    IAaveIncentivesController incentivesControllerLocal = _incentivesController;
    if (address(incentivesControllerLocal) != address(0)) {
        uint256 currentTotalSupply = _totalSupply;
        incentivesControllerLocal.handleAction(sender, currentTotalSupply, oldSenderBalance);
        if (sender != recipient) {
            incentivesControllerLocal.handleAction(recipient, currentTotalSupply, oldRecipientBalance);
        }
    }
```

ZeroLend's reward bearing tokens will call `handleAction` function but the `ZLRewardsController` contract does not have that function, it has `handleActionAfter`.

So basically, ZeroLend's token implementation and Radiant Capital's token implementations are different, but the `ZLRewardsController` is directly implemented from Radiant Capital. The contract will not work with the core protocol itself.

Note: All of these implementation contracts of Radiant Capital and ZeroLend can also be confirmed with etherscan and zksync explorer.

Furthermore I provided only "AToken" implementations of ZeroLend in this submission with the assumption of those tokens will be reward bearing tokens. But, I want to point out that other tokens in this protocol like "ZeroToken" or "vestedToken" etc. also do not invoke `handleActionAfter` at all.

Recommendation: I would recommend updating the hook in the `ZLRewardsController` contract.

3.1.3 A malicious user can inflate his voting power via `merge()`

Submitted by elhaj

Severity: High Risk

Context: ZeroLocker.sol#L711

Description: The `merge` function in the `ZeroLocker` contract is intended to allow users to consolidate their stakes by merging multiple NFTs into one. However, this function can be exploited to artificially inflate the voting power of an NFT, which can then be used to manipulate ongoing governance proposals.

An external contract will call the function `balanceOfNFT` to get the voting power for a specific `tokenId`, when a user attempt to vote with this token:

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

function _balanceOfNFT(uint256 _tokenId, uint256 _t) internal view returns (uint256) {
    uint256 _epoch = userPointEpoch[_tokenId];
    if (_epoch == 0) {
        return 0;
    } else {
        Point memory lastPoint = _userPointHistory[_tokenId][_epoch];
        lastPoint.bias -= lastPoint.slope * int128(int256(_t) - int256(lastPoint.ts));
        if (lastPoint.bias < 0) {
        lastPoint.bias = 0;
        }
        return uint256(int256(lastPoint.bias));
    }
}
```

A malicious user with multiple NFTs locked in the `ZeroLocker` contract can execute a series of merges to move their voting power into a multiple NFTs. By merging a larger-staked NFT into a smaller one, the user can increase the `bias` of the smaller NFT. The user can then use this newly merged NFT to vote and pass a malicious proposal, effectively using the same stake to vote multiple times. The process can be repeated by merging the inflated NFT into another small-staked NFT, further increasing its `bias` and using it to vote again.which can be done in the same transaction.

Proof of concept:

1. Bob creates multiple locks in the `ZeroLocker` contract, with one major lock holding a significant amount of `zeroToken` and several smaller locks with minimal amounts.
2. After some time, Bob begins the exploitation process by voting with the major lock.
3. Bob then merges the major lock into one of the smaller locks using the `merge` function, which increases the `bias` of the smaller lock due to the added tokens from the major lock.
4. With the inflated `bias` of the newly merged lock, Bob votes again, effectively using the same tokens to increase his voting power.
5. Bob repeats this process, merging the inflated lock with another small lock and voting again, further increasing his voting power each time.

This vulnerability allows a single user to have an outsized influence on the outcome of governance decisions.

Check this code implementation that shows how Bob could inflate his voting power by merging:

```
contract setup is Test {
    uint constant supply = 1000000000 ether;
    BonusPool bonus;
    VestedZeroLend vestToken;
    VestedZeroLend zeroToken;
    ZeroLocker locker;
    FeeDistributor feeDistributor;
    StakingEmissions emissions;
    StreamedVesting vest;

    function setUp() public virtual {
        // deploy zero token :
        zeroToken = new VestedZeroLend();
        // deploy vest token :
        vestToken = new VestedZeroLend();
        // deploy locker :
        locker = new ZeroLocker();
        locker.initialize(address(zeroToken));
        // deploy feedistributer :
        feeDistributor = new FeeDistributor();
        feeDistributor.initialize(address(locker), address(vestToken));
        // deploy emissions :
        emissions = new StakingEmissions();
        emissions.initialize(feeDistributor, vestToken, 100_000 ether);
        // deploy vest :
        vest = new StreamedVesting();
```

```
        // deploy bonus pool :
        bonus = new BonusPool(zeroToken, address(vest));
        vest.initialize(zeroToken, IERC20Burnable(address(vestToken)), locker, bonus);
        // initial balances :
        zeroToken.transfer(address(bonus), 5 * supply);
        zeroToken.transfer(address(vest), 55 * supply);
        vestToken.transfer(address(emissions), 50 * supply);
        vestToken.addwhitelist(address(emissions), true);
        vestToken.addwhitelist(address(feeDistributor), true);
        // disable whitelist from the zero token:
        zeroToken.toggleWhitelist(false, false);
        // start vesting and emissions :
        skip(3 days);
        vest.start();
        emissions.start();
    }

        address bob = makeAddr("bob");
        uint votingPower;
        // simulate the voting behavior :

    function vote(uint tokenId) public {
        // this calls the locker contract to get the voting power of the token id :
        votingPower += locker.balanceOfNFT(tokenId);
    }

    function test_votingPower() public {
        zeroToken.transfer(bob, 10000 ether);
        vm.startPrank(bob);
        zeroToken.approve(address(locker), 1000 ether);
        //bob create one normal lock, and 9 tiny  :
        uint[11] memory tokenIds;
        for (uint i; i < 10; i++) {
            if (i == 0) {
            // the first lock will be the major one with the biggest amount that will be moved over others to
↪   inflate the voting power:
            tokenIds[i] = locker.createLock(100 ether, 20 weeks);
            continue;
            }
            // tiny locks starting from index 2 => 10:
            tokenIds[i] = locker.createLock(100, 20 weeks);
        }
        skip(3 weeks);// just skip 3 weeks
        // catch the real voting power if bob not malicious this will be his real voting power :
        for (uint i; i < 9; i++) {
            // uninflated voting power :
            vote(tokenIds[i]);
        }
        uint uninflatedVotingPower = votingPower; // catch the real voting power
        // reset the voting power to calculate the inflated voting power when bob malicious:
        votingPower = 0;
        for (uint i; i < 9; i++) {
            // vote with majore lock:
            vote(tokenIds[i]);
            locker.merge(tokenIds[i], tokenIds[i + 1]);
        }
        // compare the inflated voting power ,and the actual voting power :
        console.log("inflated voting power    => ", votingPower);
        console.log("the real voting power    => ", uninflatedVotingPower);
        console.log("the inflated amount      => ", votingPower - uninflatedVotingPower);
    }
}
```

Output:

```
[PASS] test_votingPower() (gas: 4794558)
Logs:
    inflated voting power    =>  71506842180354954054
    the real voting power    =>  7945204686706106006
    the inflated amount      =>  63561637493648848048

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.75ms
```

The vulnerability enables a user to multiply their vote unfairly, potentially tipping the scales in favor of harmful proposals and disrupting fair governance.

Recommendation: Update the `merge` function to record a change in the `ownershipChange` mapping for the `to` NFT:

```
  function merge(uint256 _from, uint256 _to) external override {
      // prev code .....
      _depositFor(_to, value0, end, _locked1, DepositType.MERGE_TYPE);
+     ownershipChange[_to] = block.timestamp;
  }
```

3.1.4 DoS in `distribute()`

Submitted by bin2chen, also found by 0xmuxyz, osmanozdemir1, Sujith Somraaj, Ch301, 3agle, wafflemakr and shaka

Severity: High Risk

Context: StakingEmissions.sol#L57

Currently, `StakingEmissions.distribute()` is triggered once every `1 week` to transfer `VestedZeroLend` to `feeDistributor`, and then users claim from `feeDistributor`:

```
function _distribute() internal checkEpoch whenNotPaused {
    feeDistributor.checkpointToken();
    feeDistributor.checkpointTotalSupply();
    token.transfer(address(feeDistributor), amtPerEpoch);
}
```

The problem is that `feeDistributor.checkpointToken()` has a frequency limit:

```
function checkpointToken() external override {
    require(
        ((block.timestamp > lastTokenTime + TOKEN_CHECKPOINT_DEADLINE)),
        "cant checkpoint now"
    );
    _checkpointToken(block.timestamp);
}
```

The execution interval needs to be `1 day`, otherwise the call will `revert`. This may indefinitely prevent the execution of `StakingEmissions.distribute()` in the following two situations:

1. Malicious users front-run `StakingEmissions.distribute()`, preemptively calling `feeDistributor.checkpointToken();`.
2. Innocent users executing `FeeDistributor.claim(id)` will also trigger `feeDistributor.checkpointToken()`, causing `StakingEmissions.distribute()` to fail.

The distribution of rewards may be maliciously or unintentionally blocked, preventing normal distribution, possibly delaying until next week, or even longer.

Recommendation: Modify `checkpointToken()`:

```
  function checkpointToken() external override {
-     require(
-         ((block.timestamp > lastTokenTime + TOKEN_CHECKPOINT_DEADLINE)),
-         "cant checkpoint now"
-     );
-     _checkpointToken(block.timestamp);
+     if ((block.timestamp > lastTokenTime + TOKEN_CHECKPOINT_DEADLINE)) {
+         _checkpointToken(block.timestamp);
+     }
  }
```

3.1.5 Incorrect reward distribution when `t == roundedtimestamp` in `feedistributor.sol` contract

Submitted by 0xarno, also found by wafflemakr

Severity: High Risk

Context: FeeDistributor.sol#L156

Description: The `FeeDistributor` contract has a vulnerability that leads to incorrect reward distribution when a transaction occurs at the beginning of an epoch. The issue is related to the `_checkpointTotalSupply` function, where the reward calculation may be inaccurate when the timestamp exactly matches the rounded timestamp. This can result in incorrect reward amounts for users. FeeDistributor distributes newly minted tokens to users who lock the tokens in ZeroLocker. The FeeDistributor stores the supply in the public `veSupply` mapping. The FeeDistributor `_checkpointTotalSupply` function iterates from the last updated time until the latest epoch time, fetches totalSupply from ZeroLocker, and saves it.

The impact of this vulnerability is that users may receive incorrect rewards when interacting with the FeeDistributor contract. Specifically, when a transaction occurs at the beginning of an epoch and the timestamp matches the rounded timestamp, the reward calculation for subsequent transactions will be inaccurate.

Proof of concept: Assume the following scenario when a transaction is executed at the beginning of an epoch:

1. Alice locks 100 tokens, and the supply of Zero increases to 100.
2. Bob calls the `checkpointTotalSupply`. The FeeDistributor saves the totalSupply as 100.
3. Bob locks 300 tokens, and the supply of Zero increases to 400.
4. After some time, Bob claims the reward. The reward is calculated by `totalReward * balance / supply`. However, due to the vulnerability, Bob gets `reward = 3` instead of the expected `reward = 3/4`.

The invariance of the week-bound total supply is broken, as described in FeeDistributor.sol#L39.

Note: This kind of vulnerability is common in similar type of codebases.

Recommendation: It is recommended to update the `_checkpointTotalSupply` function as follows:

```
  function _checkpointTotalSupply(uint256 timestamp) internal {
      uint256 t = timeCursor;
      uint256 roundedTimestamp = (timestamp / WEEK) * WEEK;

      locker.checkpoint();

      for (uint256 index = 0; index < 20; index++) {
-         if (t > roundedTimestamp) {
+         if (t >= roundedTimestamp) {
              break;
          } else {
              uint256 epoch = _findTimestampEpoch(t);
              IZeroLocker.Point memory pt = locker.pointHistory(epoch);

              int128 dt = 0;

              if (t > pt.ts) dt = int128(uint128(t - pt.ts));

              veSupply[t] = Math.max(uint128(pt.bias - pt.slope * dt), 0);
          }

          t += WEEK;
      }

      timeCursor = t;
  }
```

3.1.6 Missing accesscontrol in `afterlockupdate`

Submitted by 0xarno, also found by erictee, osmanozdemir1, Sujith Somraaj, Chinmay Farkya, ladboy233 and Naveen Kumar J - 1nc0gn170

Severity: High Risk

Context: ZLRewardsController.sol#L588-L590

Description: The `afterLockUpdate` function in the provided `ZLRewardsController.sol` contract is intended to update a user's registered balance after locking, unlocking, or modifying a lock. However, a critical security vulnerability arises from the absence of sufficient access control mechanisms in this function. Without proper restrictions, any user, regardless of their interaction with the locking mechanism, can call `afterLockUpdate` to update their balance, potentially manipulating their eligibility for rewards.

The lack of access control in `afterLockUpdate` has several severe implications:

1. **Unauthorized Balance Manipulation:** Any user can call `afterLockUpdate` to update their balance, even if they haven't engaged in any lock-related activities. This loophole allows for potential manipulation of reward eligibility and distribution.
2. **Distortion of Reward Distribution:** The vulnerability can lead to an unfair distribution of rewards. Users who haven't locked any tokens or aren't entitled to rewards could falsely inflate their balance, thereby claiming rewards that they aren't entitled to.

Proof of concept: The vulnerability is evident in the implementation of `afterLockUpdate`:

1. The function is meant to be called post-locking activities like creating locks, unlocking, or increasing amounts through the `ZeroLocker` contract.
2. However, it lacks any checks to ensure that it is only called by these locking mechanisms (like `ZeroLocker`).
3. As a result, any user can call this function directly, updating their balance and potentially becoming eligible for rewards without actually locking any tokens.
4. It becomes evident after looking at the Natspec

Recommendation: Consider introducing access control checks in the `afterLockUpdate` function to ensure that it can only be called by authorized contracts (like `ZeroLocker`). This can be achieved by using modifiers that restrict function calls to known, trusted contracts involved in the locking process:

```
modifier onlyZeroLocker() {
    require(msg.sender == address(locker), "Unauthorized: caller is not ZeroLocker");
    _;
}
```

```
- function afterLockUpdate(address _user) external {
+ function afterLockUpdate(address _user) external onlyZeroLocker {
        _updateRegisteredBalance(_user);
    }
```

3.2 Medium Risk

3.2.1 Incorrect proxy implementation by declaring initial values in storage

Submitted by giraffe0x, also found by Saaj and wafflemakr

Severity: Medium Risk

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

Description: `StreamedVesting.sol` has two storage variables (`dead` and `duration`) that are declared with initial values and are not updated in `initialize()`. As an upgradeable contract, this would lead to critical errors when interacted with by a proxy contract:

```
address public dead = address(0xdead);
uint256 public duration = 3 * 30 days; // 3 months vesting
```

These values will be set when the contract is deployed, but they won't be stored in the proxy's state. If you do delegeatecall with a proxy, it expects `duration` to be 3 * 30 days, but it won't find this value in the proxy's state. Instead it will be the default value for its type which is 0 for uint256.

This is why it's important to initialize state variables in the initialize function found within upgradeable contracts to ensure that the initial values are correctly set in the proxy's storage.

There's no impact for `dead` as it defaults to `address(0)`. However, for `duration`, it has significant impact as there would effectively be zero vesting duration as `penalty()` will return 0% (i.e. no penalty):

```
function penalty(
    uint256 startTime,
    uint256 nowTime
) public view returns (uint256) {
    // After vesting is over, then penalty is 0%
    if (nowTime > startTime + duration) return 0;
```

Recommendation: Initialize both variables in the `initialize` function.

3.2.2 Possible underflow of `uint128`

Submitted by giraffe0x, also found by elhaj, Ch301, pauleth, Obin, 0xarno and 3agle

Severity: Medium Risk

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

`FeeDistributor.sol` is a direct fork of Curve's FeeDistributor.vy:

```
// Zerolend
function _checkpointTotalSupply(uint256 timestamp) internal {
    // ...

    veSupply[t] = Math.max(uint128(pt.bias - pt.slope * dt), 0); //@audit
}

function _claim(
        uint256 nftId,
        uint256 _lastTokenTime
    ) internal returns (uint256) {
    // ...

    uint256 balanceOf = Math.max(
        uint128(oldUserPoint.bias - dt * oldUserPoint.slope),
        0
    );
}
```

The implementation by Zerolend does a conversion to uint128 first before doing `Math.max` while Curve does max first before conversion.

```
// Curve
self.ve_supply[t] = convert(max(pt.bias - pt.slope * dt, 0), uint256)

balance_of: uint256 = convert(max(old_user_point.bias - dt * old_user_point.slope, 0), uint256)
```

The danger of doing a conversion first is that `(bias - slope * dt)` which are all signed integers has a possibility of being a negative result (i.e. less than 0). But because it is converted to a uint128 first, it will result in a very large uint128 (underflow):

```
Math.max(uint128(-1), 0) // 340282366920938463463374607431768211455
```

Recommendation: Do max comparison before converting to `uint128`:

```
veSupply[t] = uint128(int128(Math.maxInt((pt.bias - pt.slope * dt), 0)));

uint256 balanceOf = uint128(int128(Math.maxInt((oldUserPoint.bias - dt * oldUserPoint.slope),0)));
```

3.2.3 `vestedzerolend` whitelist logic is incorrectly implemented

Submitted by StErMi, also found by tnch

Severity: Medium Risk

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

Description: The whitelist and blacklist logic is implemented inside the `_afterTokenTransfer` that is called when token are minted, burned or transferred from an account to another:

```
function _afterTokenTransfer(
    address from,
    address to,
    uint256
) internal virtual override {
    if (enableWhitelist) {
        require(whitelist[from] || whitelist[to], "!whitelist");
    }

    if (enableBlacklist) {
        require(!blacklist[to] && !blacklist[from], "blacklist");
    }
}
```

Like for the `enableBlacklist` also the `enableWhitelist` should revert if both of the users are not whitelisted. Without this logic, the contract is allowing:

- A whitelisted sender to send tokens to a **non-whitelisted** receiver.
- A **non-whitelisted** sender to send tokens to a whitelisted receiver.

Recommendation: Change the check inside `_afterTokenTransfer` to revert if both the users are not whitelisted.

```
  function _afterTokenTransfer(
      address from,
      address to,
      uint256
  ) internal virtual override {
      if (enableWhitelist) {
-         require(whitelist[from] || whitelist[to], "!whitelist");
+         require(whitelist[from] && whitelist[to], "!whitelist");
      }

      if (enableBlacklist) {
          require(!blacklist[to] && !blacklist[from], "blacklist");
      }
  }
```

3.2.4 Zerolend contract will mint 100B tokens each time is deployed on a new chain

Submitted by StErMi

Severity: Medium Risk

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

Description: The current implementation of the `ZeroLend` token is minting 100B tokens each time it is deployed on a new chain.

Moreover, the `owner` of the contract is allowed to mint additional amounts of such token by calling the `mint` function in those chains.

The `ZeroLend` tokens should be minted only **once**, or at least only in the main chain where such a contract is deployed. When deployed on the other chains it should not be possible to mint tokens but just transfer them.

Recommendation: There are two options that can be followed:

1. Have two different contract logics depending if the contract is deployed on the main net or on another chain. The contract on the other chain should not be allowed to mint additional tokens.
2. Add an additional `_initialMintAmount` input parameter to the constructor and mint tokens only if it's greater than zero. When deployed on other chains, such parameter should be equal to zero. This option does not completely solve the problem introduced by the `mint(...)` function that still would allow the contract's owner to mistakenly mint tokens on other chains.

3.2.5 `streamedvesting` denies bonus when the lock is for 4 years and bonuspool has not enough to cover it

Submitted by StErMi, also found by XDZIBECX, osmanozdemir1, erictee, nethoxa, tnch, pauleth, bLnk and Naveen Kumar J - 1nc0gn170

Severity: Medium Risk

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

Description: From what we can deduce from the `StreamedVesting` usage, a user will call `createVest` or `createVestFor` to burn their `VestedZeroLend` tokens and create a vest "*record*".

At this point, the user can claim the `ZeroLend` tokens early by paying a penalty, wait for the vest duration period (3 months) and claim all the `ZeroLend` tokens that have been vested (equal to the amount of `VestedZeroLend` that has been burned) or stake those vested amount for 4 years to get a 20% (default value) bonus on the vested amount.

When `StreamedVesting.stakeTo4Year` is called, it will check if the bonus pool has at least the `bonusAmount` of `ZeroLend` token to be pulled (as a bonus). If it has not enough, it will skip such logic but continue to create the lock on the `ZeroLocker` contract:

```
function stakeTo4Year(uint256 id) external whenNotPaused {
    VestInfo memory vest = vests[id];
    require(msg.sender == vest.who, "not owner");

    uint256 lockAmount = vest.amount - vest.claimed;

    // update the lock as fully claimed
    vest.claimed = vest.amount;
    vests[id] = vest;

    // check if we can give a 20% bonus for 4 year staking
    uint256 bonusAmount = bonusPool.calculateBonus(lockAmount);
    if (underlying.balanceOf(address(bonusPool)) >= bonusAmount) {
        underlying.transferFrom(address(bonusPool), address(this), bonusAmount);
        lockAmount += bonusAmount;
    }

    // create a 4 year lock for the user
    locker.createLockFor(lockAmount, 86400 * 365 * 4, msg.sender);
}
```

If the bonus pool has not enough balance to cover the bonus, the user won't get any bonus at all, even if the balance of the pool is greater than zero. Because of this, the user will stake for 4 years their vested token without taking any bonus, even if it would not be equal to the one that he would deserve.

Recommendation: The `stakeTo4Year` function should check the bonus pool balance and the `bonusAmount` should be equal to `min(bonusPool.calculateBonus(lockAmount), underlying.balanceOf(address(bonusPool))`

By doing this, the user will at least gain the remaining some bonus, even if it's not equal to the full amount he deserves.

3.2.6 `_newrewards` does not take endrewardtime into account, which may lead to incorrect allocation of rewards

Submitted by cccz, also found by bin2chen and Ch301

Severity: Medium Risk

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

Description: `endRewardTime` represents the end time of the current emission of rewards, e.g. if the current time is `t0`, `rewardsPerSecond = 20`, `availableRewards = 8000`, then `endRewardTime = 8000/20 + t0 = t400`.

`endRewardTime` is not always is not always up-to-date, it is only updated if it has been more than `updateCadence` since the last update:

```
function endRewardTime() public returns (uint256) {
    if (
        endingTime.lastUpdatedTime + endingTime.updateCadence >
        block.timestamp
    ) {
        return endingTime.estimatedTime;
    }

    uint256 unclaimedRewards = availableRewards();
    uint256 extra = 0;
    uint256 length = poolLength();
    for (uint256 i; i < length; ) {
        PoolInfo storage pool = poolInfo[registeredTokens[i]];

        if (pool.lastRewardTime > lastAllPoolUpdate) {
            extra +=
                ((pool.lastRewardTime - lastAllPoolUpdate) *
                    pool.allocPoint *
                    rewardsPerSecond) /
                totalAllocPoint;
        }
        unchecked {
            i++;
        }
    }
    endingTime.lastUpdatedTime = block.timestamp;

    if (rewardsPerSecond == 0) {
        endingTime.estimatedTime = type(uint256).max;
        return type(uint256).max;
    } else {
        uint256 newEndTime = (unclaimedRewards + extra) /
            rewardsPerSecond +
            lastAllPoolUpdate;
        endingTime.estimatedTime = newEndTime;
        return newEndTime;
    }
}
```

In `_updatePool`, if the current time is greater than `endReward`, it will set `lastRewardTime` to `endReward`, however, in `_newRewards`, it still uses the current time to calculate the rewards, which may result in incorrect rewards allocation.

```
function _updatePool(
    PoolInfo storage pool,
    uint256 _totalAllocPoint
) internal {
    uint256 timestamp = block.timestamp;
    uint256 endReward = endRewardTime();
    if (endReward <= timestamp) {
        timestamp = endReward;
    }
    if (timestamp <= pool.lastRewardTime) {
        return;
    }

    (uint256 reward, uint256 newAccRewardPerShare) = _newRewards(
        pool,
        _totalAllocPoint
    );
    accountedRewards = accountedRewards + reward;
    pool.accRewardPerShare = pool.accRewardPerShare + newAccRewardPerShare;
    pool.lastRewardTime = timestamp;
}
```

Consider the following scenario:

- `PoolA`'s `allocPoint` is `100`, `totalAllocPoint = 100`, current time is `t0`, `rewardsPerSecond = 20`, `availableRewards = 3000`, `endRewardTime = t150`, `updateCadence = 200`.
- At `t150`, the owner calls `registerRewardDeposit` to make `availableRewards = 4000`, and since the time since the last update is less than `updateCadence`, `endRewardTime` is still `t150`. This calls `_massUpdatePools` to make `poolA.lastRewardTime = t150` and `accountedRewards = 3000`.
- When `_updatePool` is called at `t190`, for `PoolA`, `rawReward = (190-150) * 20 = 800`, `availableRewards = 1000`, `newReward = 800`. `lastRewardTime` for `PoolA` is actually `t190`, but since `endReward` is `t150`, `PoolA`'s `lastRewardTime` is `t150`, which results in more rewards being distributed to `PoolA`:

```
function _newRewards(
    PoolInfo memory pool,
    uint256 _totalAllocPoint
) internal view returns (uint256 newReward, uint256 newAccRewardPerShare) {
    uint256 lpSupply = pool.totalSupply;
    if (lpSupply > 0) {
        uint256 duration = block.timestamp - pool.lastRewardTime;
        uint256 rawReward = duration * rewardsPerSecond;

        uint256 rewards = availableRewards();
        if (rewards < rawReward) {
            rawReward = rewards;
        }
        newReward = (rawReward * pool.allocPoint) / _totalAllocPoint;
        newAccRewardPerShare =
            (newReward * ACC_REWARD_PRECISION) /
            lpSupply;
    }
}
```

Recommendation: Use `min(endRewardTime, block.timestamp)` in `_newRewards` to calculate the reward:

```
  function _newRewards(
      PoolInfo memory pool,
      uint256 _totalAllocPoint
  ) internal view returns (uint256 newReward, uint256 newAccRewardPerShare) {
      uint256 lpSupply = pool.totalSupply;
      if (lpSupply > 0) {
-         uint256 duration = block.timestamp - pool.lastRewardTime;
+         uint256 endReward = endRewardTime();
+         if (endReward <= timestamp) {
+             timestamp = endReward;
+         }
+         uint256 duration = timestamp - pool.lastRewardTime;
          uint256 rawReward = duration * rewardsPerSecond;

          uint256 rewards = availableRewards();
          if (rewards < rawReward) {
              rawReward = rewards;
          }
          newReward = (rawReward * pool.allocPoint) / _totalAllocPoint;
          newAccRewardPerShare =
              (newReward * ACC_REWARD_PRECISION) /
              lpSupply;
      }
  }
```

3.2.7 `addpool` doesn't always call `_massupdatepools`, which can lead to incorrect reward allocation

Submitted by cccz

Severity: Medium Risk

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

Description: When addPool is called, `totalAllocPoint` is increased:

```
function addPool(address _token, uint256 _allocPoint) external {
    if (msg.sender != poolConfigurator) revert NotAllowed();
    if (poolInfo[_token].lastRewardTime != 0) revert PoolExists();
    _updateEmissions();
    totalAllocPoint = totalAllocPoint + _allocPoint;
    registeredTokens.push(_token);
    PoolInfo storage pool = poolInfo[_token];
    pool.allocPoint = _allocPoint;
    pool.lastRewardTime = block.timestamp;
    pool.onwardIncentives = IIncentivesController(address(0));
    validRTokens[_token] = true;
}
```

And the change in `totalAllocPoint` will cause the pool's rewards to change, so `_massUpdatePools` should be called before `totalAllocPoint` changes to accrue the previous rewards:

```
function _newRewards(
    PoolInfo memory pool,
    uint256 _totalAllocPoint
) internal view returns (uint256 newReward, uint256 newAccRewardPerShare) {
    uint256 lpSupply = pool.totalSupply;
    if (lpSupply > 0) {
        uint256 duration = block.timestamp - pool.lastRewardTime;
        uint256 rawReward = duration * rewardsPerSecond;

        uint256 rewards = availableRewards();
        if (rewards < rawReward) {
            rawReward = rewards;
        }
        newReward = (rawReward * pool.allocPoint) / _totalAllocPoint;
        newAccRewardPerShare =
            (newReward * ACC_REWARD_PRECISION) /
            lpSupply;
    }
}
```

However, in `_updateEmissions`, `_massUpdatePools` is only called when `block.timestamp` is greater than

`endRewardTime`, and when `_massUpdatePools` is not called, the Pool's rewards are reduced as a result:

```
function _updateEmissions() internal {
    if (block.timestamp > endRewardTime()) {
        _massUpdatePools();
        lastRPS = rewardsPerSecond;
        rewardsPerSecond = 0;
        return;
    }
    setScheduledRewardsPerSecond();
}
```

Consider the following scenario, the current time is `t0`, `PoolA`'s `allocPoint` is `100`, `totalAllocPoint` is `100`, `availableRewards = 4000`, `rewardsPerSecond = 20`, and `endRewardTime = t200`.

At `t150`, `PoolA`'s total rewards are `20 * 150 * 100/100 = 3000`.

If the owner calls `addPool` to add `PoolB`, with `allocPoint` of `50`, since `_massUpdatePools` is not called, the total rewards for `PoolA` are `20 * 150 * 100/150 = 2000`, which is 1000 less than before.

Recommendation: Always call `_massUpdatePools` in `addPools`:

```
  function addPool(address _token, uint256 _allocPoint) external {
      if (msg.sender != poolConfigurator) revert NotAllowed();
      if (poolInfo[_token].lastRewardTime != 0) revert PoolExists();
+     _massUpdatePools();
      _updateEmissions();
      totalAllocPoint = totalAllocPoint + _allocPoint;
      registeredTokens.push(_token);
      PoolInfo storage pool = poolInfo[_token];
      pool.allocPoint = _allocPoint;
      pool.lastRewardTime = block.timestamp;
      pool.onwardIncentives = IIncentivesController(address(0));
      validRTokens[_token] = true;
  }
```

3.2.8 `supply` in the `zerolocker.sol` is incorrectly increased during `merge_type` actions

Submitted by osmanozdemir1, also found by m4rio, Chinmay Farkya, jovi.eth, Sujith Somraaj, 0x175, Naveen Kumar J - 1nc0gn170, bLnk, Ch301, ethan, elhaj, wafflemakr, 3agle and 0xarno

Severity: Medium Risk

Context: ZeroLocker.sol#L674, ZeroLocker.sol#L726

Description: Users are able to create locks, deposit tokens for already created locks, increase lock amount/time, and merge two different locks in the `ZeroLocker` contract.

The `supply` public variable in this contract tracks the deposited underlying token amounts. It is increased with every deposit and decreased with every withdraw. The issue here is that the `supply` is incremented with all types of deposits without checking the deposit type.

Here we can see the `_depositFor` function:

```
function _depositFor(
    uint256 _tokenId,
    uint256 _value,
    uint256 unlockTime,
    LockedBalance memory lockedBalance,
    DepositType depositType
) internal {
    LockedBalance memory _locked = lockedBalance;
    uint256 supplyBefore = supply;

    supply = supplyBefore + _value; //@audit-issue MERGE should not increase the supply
    LockedBalance memory oldLocked;
    (oldLocked.amount, oldLocked.end) = (_locked.amount, _locked.end);

    // skipped for brevity.

    address from = msg.sender;
    if (_value != 0 && depositType != DepositType.MERGE_TYPE) {
        assert(underlying.transferFrom(from, address(this), _value)); //@audit Underlying token is not
↪   transferred if it is MERGE_TYPE
    }

    emit Deposit(
        from,
        _tokenId,
        _value,
        _locked.end,
        depositType,
        block.timestamp
    );
    emit Supply(supplyBefore, supplyBefore + _value);
}
```

`MERGE_TYPE` deposits do not transfer underlying tokens, **do not change the supply**, but the supply state variable is incorrectly incremented.

Down below, you can see that the `_depositFor` function is called with the amount of `locked0` in the `merge` function:

```
function merge(uint256 _from, uint256 _to) external override {
    require(_from != _to, "same nft");
    require(_isApprovedOrOwner(msg.sender, _from), "from not approved");
    require(_isApprovedOrOwner(msg.sender, _to), "to not approved");

    LockedBalance memory _locked0 = locked[_from];
    LockedBalance memory _locked1 = locked[_to];
    uint256 value0 = uint256(int256(_locked0.amount));
    uint256 end = _locked0.end >= _locked1.end
        ? _locked0.end
        : _locked1.end;

    locked[_from] = LockedBalance(0, 0, 0);
    _checkpoint(_from, _locked0, LockedBalance(0, 0, 0));
    _burn(_from);
    _depositFor(_to, value0, end, _locked1, DepositType.MERGE_TYPE); //@audit-issue value0 is the amount of
↪   the locked0. The supply will be increased as much as value0. This action should not increase the supply.
}
```

Every `merge` call will wrongly update the state. There will be a significant mismatch between the actual deposited underlying tokens and the `supply`.

Likelihood is high. It will certainly happen. This might be with regular user interactions or a malicious user might intentionally merge multiple positions to arbitrarily increase the supply.

The impact is low in the current codebase since the supply is not used in important calculations. However, it creates a broken invariant, emits `Supply` event with incorrect values, and the `totalSupplyWithoutDecay` function will return much higher TVL values. Besides, this may cause serious integration problems and result in incorrect calculations if this state variable is used in the core-contracts of this protocol in the future or other protocols building on top of this one.

Proof of concept:

- Create an empty file in the `test` folder and name it `something.ts`.

- Copy and paste the snippet into the newly created file.
- Run it with `npx hardhat test --grep "incorrectly increases the supply during merge"`:

```
import { expect } from "chai";
import { ethers } from "hardhat";
import {
    loadFixture,
    time,
} from "@nomicfoundation/hardhat-toolbox/network-helpers";

import { e18, deployCore as fixture } from "./fixtures/core";

describe("ZeroLocker", () => {
    it("incorrectly increases the supply during merge", async function () {
        const {
          locker,
          token,
          otherAccount: user,
        } = await loadFixture(fixture);

        const AMOUNT = e18 * 1000000n;

        // Send some tokens to user and approve.
        await token.transfer(user.address, AMOUNT * 10n);
        await token
          .connect(user)
          .approve(locker.target.toString(), AMOUNT * 10n);

        // User creates a lock for 4 years
        await locker.connect(user).createLock(AMOUNT, 86400 * 365 * 4);

        // "suppy" is equal to AMOUNT
        // underlying balance of the contract is equal to AMOUNT
        const supplyAfterFirstLock = await locker.supply();
        const balanceAfterFirstLock = await token.balanceOf(locker);
        expect(supplyAfterFirstLock).eq(AMOUNT);
        expect(balanceAfterFirstLock).eq(AMOUNT);

        // User creates another lock for 2 years
        await locker.connect(user).createLock(AMOUNT, 86400 * 365 * 2);

        // "supply" is (2 * AMOUNT)
        // underlying balance of the contract is equal to (2 * AMOUNT)
        const supplyAfterSecondLock = await locker.supply();
        const balanceAfterSecondLock = await token.balanceOf(locker);
        expect(supplyAfterSecondLock).eq(AMOUNT * 2n);
        expect(balanceAfterSecondLock).eq(AMOUNT * 2n);

        // ----------------MERGE-----------------
        // User merges
        await locker.connect(user).merge(1, 2);

        // Supply incorrectly increased to (3 * AMOUNT)
        const supplyAfterMerge = await locker.supply();
        expect(supplyAfterMerge).eq(AMOUNT * 3n);

        // But the underlying balance is still (2 * AMOUNT)
        const balanceAfterMerge = await token.balanceOf(locker);
        expect(balanceAfterMerge).eq(AMOUNT * 2n);
    });
});
```

Recommendation: Check the deposit type in the `_depositFor` function before increasing the supply.

```
  function _depositFor(
      uint256 _tokenId,
      uint256 _value,
      uint256 unlockTime,
      LockedBalance memory lockedBalance,
      DepositType depositType
  ) internal {
      LockedBalance memory _locked = lockedBalance;
      uint256 supplyBefore = supply;

-     supply = supplyBefore + _value;
+     if(depositType != DepositType.MERGE_TYPE) {
+           supply = supplyBefore + _value;
+     }

      // rest of the code
```

3.2.9 `streamedvesting.claimvestearlywithpenalty()` charges much more penalties than required

Submitted by osmanozdemir1, also found by giraffe0x, StErMi, jovi.eth, elhaj and Chinmay Farkya

Severity: Medium Risk

Context: StreamedVesting.sol#L145, StreamedVesting.sol#L154

Description: Users can vest their `vestedTokens` for 3 months (90 days) and claim `underlyingTokens` using the `StreamedVesting` contract. There are two types of claiming functionality in this contract after vesting some tokens.

- `claimVest`: Users can claim proportional amount of tokens depending on claiming time **without any penalties**. e.g: Users can claim 2/3 of their tokens if they claim after 60 days.
- `claimVestEarlyWithPenalty`: Users can claim their whole vesting amount before 90 days but they pay some penalty depending on claiming time.

See the `claimVestEarlyWithPenalty`:

```
function claimVestEarlyWithPenalty(uint256 id) external whenNotPaused {
    VestInfo memory vest = vests[id];
    require(msg.sender == vest.who, "not owner");

    uint256 pendingAmt = vest.amount - vest.claimed; //@audit it caches "whole" unclaimed amount
    require(pendingAmt > 0, "no pending amount");

    // update
    vest.claimed += pendingAmt;
    vests[id] = vest;

    // send reward with penalties
    uint256 penaltyPct = penalty(vest);
    uint256 penaltyAmt = ((pendingAmt * penaltyPct) / 1e18); //@audit-issue Applies penalties to whole
↪   unclaimed amount.
    underlying.transfer(msg.sender, pendingAmt - penaltyAmt);
    underlying.transfer(dead, penaltyAmt);

    emit TokensReleased(msg.sender, id, penaltyAmt);
    emit PenaltyCharged(msg.sender, id, penaltyAmt, penaltyPct);
}
```

This function caches unclaimed amount, gets the penalty percentage, and **applies this penalty percentage to the whole unclaimed amount**.

Unfortunately, this is wrong. Because if a user creates vest with 300 tokens and decides to claim early in 60th day, he/she should get 200 tokens without any penalty for two months of vesting and the penalty should only be applied to the remaining 100 tokens.

However, as we can see above, the function applies the penalty to the whole 300 tokens if the user didn't claim before.

This also creates inconsistent behaviours. A user will pay much higher penalties if that user only calls

`claimVestEarlyWithPenalty`, compared to calling the `claimVest` first and then calling `claimVestEarlyWithPenalty`.

Likelihood is high since a regular user can not know this inconsistent behaviour and they will probably call only the `claimVestEarlyWithPenalty` if they want to claim early.

Impact is high since there is unfair loss of funds. Users don't get their fair share and pay much more penalties.

Proof of concept:

- Copy and paste snippet into the `StreamedVesting.ts` test file.
- Run it with `npx hardhat test --grep "different penalties"`:

```
it("Charges different penalties for an early withdrawal depending on previous claims", async function () {
    const { vesting, vestedToken, token, otherAccount, owner } = await loadFixture(
    fixture
    );

    // We'll use two different accounts performing the same actions at the same time.
    // Only difference is one of them will call only "claimVestEarlyWithPenalty", and the other will call
↪   "claimVest" + "claimVestEarlyWithPenalty".

    // Owner gives approval
    expect(await vestedToken.balanceOf(owner.address)).greaterThan(0);
    await vestedToken.approve(vesting.target.toString(), e18 * 100n);

    // Transfer some to the other account
    await vestedToken.transfer(otherAccount.address, e18 * 100n);
    expect(await vestedToken.balanceOf(otherAccount.address)).eq(e18 * 100n);

    // Other account gives approval
    await vestedToken
    .connect(otherAccount)
    .approve(vesting.target.toString(), e18 * 100n);

    // Both otherAccount and owner creates vests at the same time and same amount.
    await vesting.connect(otherAccount).createVest(e18 * 100n);
    expect(await vesting.claimable(1)).to.equal(0);
    await vesting.connect(owner).createVest(e18 * 100n);
    expect(await vesting.claimable(2)).to.equal(0);

    // Fast forward 2 months.
    await time.increase(86400 * 60);

    // Store balances before claiming vests.
    const ownerTokenBalanceBefore = await token.balanceOf(owner.address);
    const otherAccountBalanceBefore = await token.balanceOf(otherAccount.address);

    // Other account calls "claimVest" and "claimVestEarlyWithPenalty" back to back.
    // Owner calls only "claimVestEarlyWithPenalty"
    await vesting.connect(otherAccount).claimVest(1);
    await vesting.connect(otherAccount).claimVestEarlyWithPenalty("1");
    await vesting.connect(owner).claimVestEarlyWithPenalty("2");

    // Store balances after claiming vests.
    const ownerTokenBalanceAfter = await token.balanceOf(owner.address);
    const otherAccountBalanceAfter = await token.balanceOf(otherAccount.address);

    // Other account's delta is greater than the owner's delta.
    // The reason for that is the "claimVestEarlyWithPenalty" function applies penalties to all unclaimed
↪   amounts.
    const ownerDelta = ownerTokenBalanceAfter - ownerTokenBalanceBefore;
    const otherAccountDelta = otherAccountBalanceAfter - otherAccountBalanceBefore;
    expect(otherAccountDelta).greaterThan(ownerDelta);
});
```

Recommendation: One option is calling the regular `claimVest` function as a first step inside the `claimVestEarlyWithPenalty` function. However this will cause transferring tokens twice and probably cause more gas consumption.

Other option is calculating claimable amount without penalties first and applying penalties to the remaining amount. For example:

```
  // NOTE: This is just an example.

  function claimVestEarlyWithPenalty(uint256 id) external whenNotPaused {
      VestInfo memory vest = vests[id];
      require(msg.sender == vest.who, "not owner");

+     uint256 claimableWithoutPenalty = _claimable(vest);

-     uint256 pendingAmt = vest.amount - vest.claimed;
+     // The penalty will be applied only to the remaining part of funds.
+     uint256 pendingAmt = vest.amount - vest.claimed - claimableWithoutPenalty;
      require(pendingAmt > 0, "no pending amount");

      // update
-     vest.claimed += pendingAmt;
+     vest.claimed = vest.claimed + pendingAmt + claimableWithoutPenalty;
      vests[id] = vest;

      // send reward with penalties
      uint256 penaltyPct = penalty(vest);
      uint256 penaltyAmt = ((pendingAmt * penaltyPct) / 1e18);
-     underlying.transfer(msg.sender, pendingAmt - penaltyAmt);
+     // Claimable without penalty will be transferred fully. And pending will be transferred after penalty.
+     underlying.transfer(msg.sender, claimableWithoutPenalty + pendingAmt - penaltyAmt);
      underlying.transfer(dead, penaltyAmt);

      emit TokensReleased(msg.sender, id, penaltyAmt);
      emit PenaltyCharged(msg.sender, id, penaltyAmt, penaltyPct);
  }
```

3.2.10 Transferring tokens to `feedistributor` after calling `checkpointtoken` may cause incorrect distribution

Submitted by osmanozdemir1

Severity: Medium Risk

Context: FeeDistributor.sol#L63, StakingEmissions.sol#L56

Description: `StakingEmissions` contract transfers tokens to `FeeDistributor` as weekly emissions, and users can claim rewards from `FeeDistributor`. Here is the `StakingEmissions._distribute()` function:

```
function _distribute() internal checkEpoch whenNotPaused {
    feeDistributor.checkpointToken();
    feeDistributor.checkpointTotalSupply();
    token.transfer(address(feeDistributor), amtPerEpoch);
```

It calls checkpoint first, and then transfers tokens. Let's check `FeeDistributor._checkpointToken()`:

```
function _checkpointToken(uint256 timestamp) internal {
    uint256 tokenBalance = token.balanceOf(address(this)); //@audit it uses contract balance to determine how
↪   much will be distributed. But this is called before token transfer.
    uint256 toDistribute = tokenBalance - tokenLastBalance;
    tokenLastBalance = tokenBalance;

    uint256 t = lastTokenTime;
    uint256 sinceLast = timestamp - t;
    lastTokenTime = timestamp;

    // Skipped for brevity
```

As we can see above, the checkpoint action is made with the **contract balance**. However, as I mentioned above, checkpoint is called first and then the balance is increased.

We also now that the checkpoint function can not be called again for next 24 hours. So, for the next 24 hours after `StakingEmissions._distribute()` transaction, the actual balance of the `FeeDistributor` contract and the last "*checkpointed*" balance will differ, which may cause incorrect `toDistribute` amounts.

A user claiming in the first 24 hour after regular token emission may get less distributed tokens compared to another user who claims two days later even though they lock same amount of tokens in the same epoch.

Another problem with that if `StakingEmissions._distribute()` is called between 6th and 7th days of the `epoch X`, some portion of the transferred tokens for `epoch X` will be accounted for `epoch X + 1` since these tokens will be "*checkpointed*" 24 hours later and it will create unfair distribution.

Recommendation: I would recommend transferring tokens first and then calling checkpoint after the transfer.

3.2.11 `zerolend.votingpowerof()` doesn't check `ownershipchange` block of the token, makes flash nft protection redundant

Submitted by osmanozdemir1, also found by Chinmay Farkya, tnch, wafflemakr, 3agle and karanctf

Severity: Medium Risk

Context: ZeroLocker.sol#L186

Description: Users lock their underlying tokens to the `ZeroLocker` contract and get NFTs representing their lock in the protocol. It is stated that token holders get benefits, rewards and voting rights.

Voting rights of these tokens can be seen with `votingPowerOf()`. I'll come back to this function later.

Before moving forward, I would like to point out that this protocol has a mapping called `ownershipChange`, which tracks which block the token is transferred. According to developer comments, this is done for flash NFT protection. We can see it in the `_transferFrom` function:

```
// function _transferFrom
// skipped for brevity

// Set the block of ownership transfer (for Flash NFT protection)
ownershipChange[_tokenId] = block.number;
```

Now, let's check the external `balanceOfNFT` function:

```
function balanceOfNFT(
    uint256 _tokenId
) external view override returns (uint256) {
    if (ownershipChange[_tokenId] == block.number) return 0; //@audit it returns 0 if it is called in the same
↪   block with the transfer. However, this check is not present in the "votingPowerOf" function.
    return _balanceOfNFT(_tokenId, block.timestamp);
}
```

We can see that this function returns 0 if it is called in the same block with the token transfer. The reason for that is obviously to prevent buying and voting in the same block and perform flash NFT protection as the developer comments explains.

However, this check is not present in the [votingPowerOf](https://cantina.xyz/code/a83eaf73-9cbc-495f-8607-e55d4fdaf407/contracts/ZeroLocker.sol- L186) function.

```
function votingPowerOf(
    address _owner
) external view returns (uint256 _power) {
    for (uint256 index = 0; index < ownerToNFTokenCount[_owner]; index++) {
        uint256 _tokenId = ownerToNFTokenIdList[_owner][index];
        _power += _balanceOfNFT(_tokenId, block.timestamp);
    }
}
```

It will iterate all NFTs and will call the internal `_balanceOfNFT` function (which is also not have this check). Voting powers for these NFTs will be calculated as normal even in the same block with the transfer.

Therefore, there will be no flash NFT protection if the voting mechanism integration is made with the `votingPowerOf` function.

Note: I acknowledge this is a view function and the impact is limited within the current codebase in the scope. However, I assume that this function will be used for actual voting and user voting powers will be determined based on that. I submitted it as medium severity even though voting manipulations would normally be high because of that assumption.

Recommendation: `ownershipChange` of the NFT should be check for this function too similar to `balanceOfNFT` function.

```
  function votingPowerOf(
      address _owner
  ) external view returns (uint256 _power) {
      for (uint256 index = 0; index < ownerToNFTokenCount[_owner]; index++) {
          uint256 _tokenId = ownerToNFTokenIdList[_owner][index];
-         _power += _balanceOfNFT(_tokenId, block.timestamp);
+         // Only add voting power if it is not the same block.
+         if (ownershipChange[_tokenId] != block.number) {
+               _power += _balanceOfNFT(_tokenId, block.timestamp);
+         }
      }
  }
```

3.2.12 NFTs can be transferred to zero address and fees can be claimed for these tokenids

Submitted by Chinmay Farkya, also found by osmanozdemir1 and Naveen Kumar J - 1nc0gn170

Severity: Medium Risk

Context: FeeDistributor.sol#L264, ZeroLocker.sol#L191, ZeroLocker.sol#L325

Description: The comment in `_transferFrom` says that it does not allow transferring to the zero address. This gives a false impression because this is not true.

Any tokenID can in fact be transferred to zero address because the transferFrom flow does not check for it anywhere. The `_to` parameter can be `address(0)` and the transfer will happen along with all the state updates (i.e. `now address(0)` will be the owner of this tokenID and it will be in the list of tokenIDs for `ownerToNFTokenIdList[address(0)]`).

This is similar to burning an NFT position (via merge/ withdraw) because the associated voting power will not be accessible to the previous owner, and the deposited asset can never be withdrawn by the actual owner of the tokenID, after this transfer happens. But there are some differences.

There will be no checkpoint for this kind of `burn` until the lock expires, the voting power i.e. bias and slope values will keep getting considered in global point history calculations, only after that the slopechanges might get applied.

The total "*supply*" value becomes wrong and is not reduced like it should be in case of a burn. This causes wrong accounting external voting contracts (e.g. wrong quorum calculations). This can never be corrected even after the lock expires.

Fees can still be claimed via the feedistributor because it doesn't check if the `ownerOf(tokenID)` is not `address(0)` and because there has been no ending checkpoint (like the one in withdraw $\rightarrow$ `_burn` flow setting a new checkpoint with zero values). So even though the claimed fees will go to `address(0)`, claim can be called by anyone until the lock expires or the maxepoch has passed.

The underlying assets will be stuck forever in the zerolocker contract.

And the voting power of these tokenIDs held by the zero address can be used by anyone because `votingPowerOf` / `_balanceOfNFT` etc... functions do not check if the owner of `tokenID` is `address(0)`, and non-zero checkpoint exists based on which these tokenIDs will have a substantial voting power until the lock expires.

Because of so many different impacts, this is a critical severity issue.

Recommendation: Follow the comment, revert when `to` parameter is `address(0)` for accounting consistency everywhere.

3.2.13 Voting power miscalculation in `balanceofnftat` function

Submitted by elhaj

Severity: Medium Risk

Context: ZeroLocker.sol#L961

Description: The `ZeroLocker` contract serves as a foundational component in the governance of the zerolend protocol, enabling stakeholders to lock ZERO tokens and obtain voting power proportional to their stake and commitment duration. Accurate calculation of this voting power is crucial for ensuring that governance decisions and protocol changes reflect the true consensus of the community.

The `balanceOfNFTAt` function in the `ZeroLocker` contract retrieves the voting power of a specific `tokenId` at a given timestamp `_t`. This function is essential for determining voting power at past timestamps for snapshot-based voting:

```
function balanceOfNFTAt(uint256 _tokenId, uint256 _t) external view returns (uint256) {
    return _balanceOfNFT(_tokenId, _t);
}

function _balanceOfNFT(uint256 _tokenId, uint256 _t) internal view returns (uint256) {
    uint256 _epoch = userPointEpoch[_tokenId];
    if (_epoch == 0) {
        return 0;
    } else {
        Point memory lastPoint = _userPointHistory[_tokenId][_epoch];
        lastPoint.bias -= lastPoint.slope * int128(int256(_t) - int256(lastPoint.ts));
        if (lastPoint.bias < 0) {
        lastPoint.bias = 0;
        }
        return uint256(int256(lastPoint.bias));
    }
}
```

However, the function incorrectly calculates voting power for timestamps in the past. Specifically, when `_t` is before the timestamp of the user's last checkpoint (`lastPoint.ts`), the calculation improperly increases the `bias`. This results for example to users who staked after the snapshot timestamp being erroneously granted voting power, contrary to the intended logic where only tokens staked before the snapshot should count, or this could lead to users who's being staking at this time then withdraw thier lock, to not having any voting power due to thier `lastPoint.ts` decreased `bias`.

The miscalculation leads to a situation where the later a user stakes, the more their voting power is incorrectly increased, due to the subtraction of a negative number. To simplify this issue consider the following example:

- A governance vote takes a snapshot at `time = 1000` for voting eligibility.
- The current blockchain timestamp is `1500`.
- A user decides to stake `10000e18` tokens at `time = 1100`, which is after the snapshot time.
- Upon staking, the user's `userPointEpoch[tokenId]` is set to `1`.
- The user's `userHistoryPoint[tokenId][1]` records their voting power with a `slope` and `bias` based on the stake amount and lock duration.
  - The user's `slope` is calculated as follows:
    * `slope = amount of tokens locked / MAXTIME`
    * `slope = 10000e18 / (4 * 365 * 86400)  79274479959411.47`
  - The user's `bias` at the time of staking (`time = 1100`) is calculated as:
    * `bias = slope * MAXTIME` (assume he stake for 4 years)
    * `bias = 79274479959411.47 * (4 * 365 * 86400)= 10000e18`

  Now, let's calculate the user's voting power at the snapshot time (`time = 1000`), which should be zero since the user staked after the snapshot:

```
lastPoint.bias -= lastPoint.slope * int128(int256(_t) - int256(lastPoint.ts));
```

so:

  - `bias = bias - (slope * (1000 - 1100))`
  - `bias = 10000e18 - (79274479959411.47 * (1000 - 1100))`
  - `bias at time = 1000 = 10000e18 + 7927447995941147.0 = 10000.007927447996e18` (Incorrect increase due to negative multiplication)

- The function should not count the user's stake at `time = 1000` because the user staked at `time = 1100`. However, due to the subtraction of a negative number (since `1000 - 1100` is `-100`), the `bias` is incorrectly increased instead of being zero. This results in the user erroneously having a voting power of `10000.007927447996e18` at `time = 1000`, which should not be possible as the user did not have any tokens staked at that time.

  `notice` : that a malicious user can frontrun the call to this function , to stake and have a voting power.

- The second issue is that The function does not include protection against flash loan attacks like in the function `balanceOfNft` when the current `block.timestamp` is passed as `_t`. This could allow an attacker to use a flash loan to temporarily inflate their balance and manipulate the outcome of a vote.

Proof of concept: The proof of concept shows how *Bob* was able to grant more voting power than Alice even though he staked after the snapshot and in the contrary his voting power increases over time and does not decrease:

```
contract setup is Test {
    uint constant supply = 1000000000 ether;
    BonusPool bonus;
    VestedZeroLend vestToken;
    VestedZeroLend zeroToken;
    ZeroLocker locker;
    FeeDistributor feeDistributor;
    StakingEmissions emissions;
    StreamedVesting vest;

    function setUp() public virtual {
        // deploy zero token :
        zeroToken = new VestedZeroLend();
        // deploy vest token :
        vestToken = new VestedZeroLend();
        // deploy locker :
        locker = new ZeroLocker();
        locker.initialize(address(zeroToken));
        // deploy feedistributer :
        feeDistributor = new FeeDistributor();
        feeDistributor.initialize(address(locker), address(vestToken));
        // deploy emissions :
        emissions = new StakingEmissions();
        emissions.initialize(feeDistributor, vestToken, 100_000 ether);
        // deploy vest :
        vest = new StreamedVesting();
        // deploy bonus pool :
        bonus = new BonusPool(zeroToken, address(vest));
        vest.initialize(zeroToken, IERC20Burnable(address(vestToken)), locker, bonus);
        // initial balances :
        zeroToken.transfer(address(bonus), 5 * supply);
        zeroToken.transfer(address(vest), 55 * supply);
        vestToken.transfer(address(emissions), 50 * supply);
        vestToken.addwhitelist(address(emissions), true);
        vestToken.addwhitelist(address(feeDistributor), true);
        // disable whitelist from the zero token:
        zeroToken.toggleWhitelist(false, false);
        // start vesting and emissions :
        skip(3 days);
        vest.start();
        emissions.start();
    }

    function test_votingPower() public {
        // alice is a user who locked before snapshot:
        zeroToken.transfer(alice, 1000 ether);
        vm.startPrank(alice);
        zeroToken.approve(address(locker), 1000 ether);
```

```
        uint tokenId0 = locker.createLock(10 ether, 20 weeks);
        vm.stopPrank();
        // let's say 5 weeks passed , and bob staked :
        skip(5 weeks);
        // bob create a lock :
        zeroToken.transfer(bob, 1000 ether);
        vm.startPrank(bob);
        zeroToken.approve(address(locker), 1000 ether);
        uint tokenId1 = locker.createLock(10 ether, 20 weeks);
        // now lets get the voting power of alice and bob , before even bob staked :
        uint snapshot = block.timestamp - 2 weeks;
        uint bobVotingPower = locker.balanceOfNFTAt(tokenId1, snapshot);
        uint aliceVotingPower = locker.balanceOfNFTAt(tokenId0, snapshot);
        int bobBias = locker.userPointHistory(tokenId1, 1).bias; // get the bias of bob :
        assertTrue(bobVotingPower > uint(bobBias));
        assertTrue(bobVotingPower > aliceVotingPower);
    }
}
```

Impact: The miscalculation in the `balanceOfNFTAt` function can lead to two issues:

1. Users who stake after a snapshot can erroneously be granted voting power for that snapshot, allowing them to influence the outcome of a vote in which they should not be eligible to participate.
2. If the current `block.timestamp` is used as the timestamp `_t`, the lack of flash loan protection in the function may allow an attacker to exploit this vulnerability for double. This could further compromise the fairness and security of the voting system.

Recommendation: Implement a logic that do binary search for the last `epoch` of the `tokenId` in question, then calculate the voting power depends on the bias and the slop in this `epoch`.

3.2.14 `initiallastpoint` mismanagement compromises epoch accuracy and snapshot governance

Submitted by elhaj

Severity: Medium Risk

Context: ZeroLocker.sol#L558, ZeroLocker.sol#L973

Description: The `ZeroLocker` contract is designed to track and calculate voting power for governance purposes based on users' locked tokens. A crucial aspect of this calculation involves determining the voting power at specific `block heights`, which is necessary for snapshot-based voting.

```
function balanceOfAtNFT(uint256 _tokenId, uint256 _block) external view returns (uint256) {
    return _balanceOfAtNFT(_tokenId, _block);
}
```

The contract maintains a history of checkpoints, each with a `Point` struct that records the voting power (`bias` and `slope`) and time (`ts`) and the block number (`blk`) at that time.

In the checkPoint function, the contract uses an `initialLastPoint` variable to store the state of the last checkpoint and calculate the block number for new epochs.

```
function _checkpoint(uint256 _tokenId, LockedBalance memory oldLocked, LockedBalance memory newLocked)
↪  internal {
    // prev code ...
    Point memory lastPoint = Point({ bias: 0, slope: 0, ts: block.timestamp, blk: block.number });
    if (_epoch > 0) {
        lastPoint = _pointHistory[_epoch];
    }
    uint256 lastCheckpoint = lastPoint.ts;
    // initialLastPoint is used for extrapolation to calculate block number
    // (approximately, for *At methods) and save them
    // as we cannot figure that out exactly from inside the contract
    // @audit-issue : this initialLastPoint get updated when ever lastPoint get updated
    Point memory initialLastPoint = lastPoint;
    uint256 blockSlope = 0; // dblock/dt
    if (block.timestamp > lastPoint.ts) {
        blockSlope = (MULTIPLIER * (block.number - lastPoint.blk)) / (block.timestamp - lastPoint.ts);
    }
    // more code ...
}
```

However, due to Solidity's reference semantics for struct types in memory, the line `Point memory initialLastPoint = lastPoint;` does not create an independent copy of `lastPoint`. Instead, `initialLastPoint` and `lastPoint` *points to the same data in memory*. As a result, any subsequent modifications to `lastPoint` are also reflected in `initialLastPoint`.Solidity documentation states:

> Assigning a struct to a local variable or a function's return variable creates an independent copy if and only if the type of the variable is a storage reference..

This behavior is contrary to the intended use of `initialLastPoint`, which should remain unchanged to accurately calculate the block number for each epoch.

Now, since `initialLastPoint` is a memory variable that changes when ever `lastPoint` changes, this will lead to the following miscalculation:

```
lastPoint.ts = tI;
lastPoint.blk = initialLastPoint.blk + (blockSlope * (tI - initialLastPoint.ts)) / MULTIPLIER;
```

Here, `(tI - initialLastPoint.ts)` is expected to be the time difference since the last checkpoint, but due to the shared reference, it is always zero, because `tI` will always equal `initialLastPoint.ts`. Consequently, `lastPoint.blk` remains the same as `initialLastPoint.blk`, causing all epochs to incorrectly reference the block number of the first checkpoint.

Proof of Concept: This proof of concept shows that when assigning a struct to another struct in memory, when ever one of the structs changes, the other will also changes since they point to the same data in memory:

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
import "forge-std/Test.sol";
contract POCs is Test{

    struct Point {
        int128 bias;
        int128 slope; // -  -dweight / dt
        uint256 ts;
        uint256 blk; // block
    }

    function test_unapdatedStruct() public {
        // initiate a point (assume this is the last point) :
        Point memory lastPoint = Point({ bias: 1000, slope: 1, ts: block.timestamp, blk: block.number });
        uint256 lastCheckpoint = lastPoint.ts;
        // initialLastPoint is used for extrapolation to calculate block number
        // (approximately, for *At methods) and save them
        // as we cannot figure that out exactly from inside the contract
        // @audit-issue : this initialLastPoint is actualy get updated when ever lastPoint get updated
        Point memory initialLastPoint = lastPoint;
        // now let's update the lastPoint struct :
        lastPoint.bias = 200;
        lastPoint.slope = 10;
        // this will also update the initialLastPoint struct :
        assertEq(initialLastPoint.bias, 200);
        assertEq(initialLastPoint.bias, lastPoint.bias);
    }
}
```

Impact: The integrity of past governance votes is compromised, as the voting power used to determine outcomes not correspond to the actual block. As an example this call will always revert:

```
/// @notice Calculate total voting power at some point in the past
/// @param _block Block to calculate the total voting power at
/// @return Total voting power at `_block`
function totalSupplyAt(uint256 _block) external view override returns (uint256) {
    // some code ....
        dt = ((_block - point.blk) * (pointNext.ts - point.ts)) / (pointNext.blk - point.blk); // @audit div
↪   by zero since pointNext.blk  == point.blk
    // ...
    }
```

The contract's ability to provide a verifiable record of voting power over time is undermined.

Recommendations: Initiate a new `initialLastPoint` with the same values of `lastPoint`:

```
  function _checkpoint(uint256 _tokenId, LockedBalance memory oldLocked, LockedBalance memory newLocked)
↪  internal {
      // prev code ...
      Point memory lastPoint = Point({ bias: 0, slope: 0, ts: block.timestamp, blk: block.number });
      if (_epoch > 0) {
          lastPoint = _pointHistory[_epoch];
      }
      uint256 lastCheckpoint = lastPoint.ts;
      // initialLastPoint is used for extrapolation to calculate block number
      // (approximately, for *At methods) and save them
      // as we cannot figure that out exactly from inside the contract
+     Point memory initialLastPoint = Point(lastPoint.bias, lastPoint.slope, lastPoint.ts, lastPoint.blk);
-     Point memory initialLastPoint = lastPoint;
      uint256 blockSlope = 0; // dblock/dt
      if (block.timestamp > lastPoint.ts) {
          blockSlope = (MULTIPLIER * (block.number - lastPoint.blk)) / (block.timestamp - lastPoint.ts);
      }
      // more code ...
  }
```

3.2.15 Allowing merge of expired locks may prevent reward claiming and break some invariants

Submitted by elhaj, also found by 0xTheBlackPanther, Naveen Kumar J - 1nc0gn170, bin2chen, ethan and 0xarno

Severity: Medium Risk

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

Description: The `merge` function in the `ZeroLocker` contract is designed to combine the stakes of two NFTs, transferring the locked tokens from one NFT (`from` tokenId) to another (`to` tokenId).

The `merge` function updates the `locked[_to].end` to the end time of `_from` if it's the biggest one (`locked[_from]` more then `locked[_to].end`), *but it does not properly handle the scenario where any of this two tokenIds already expired*. allowing the merge of an expired `nftId` (where the lock's end time has passed) with an `nftId` that's not expired where the `to` is the expired one may prevent users from getting any rewards when they try to claim the rewards for this merged `nftId` in some sinarios.consider the follwing example:

1. A user's lock expires and they did not claim their accumulated rewards immediately (at least one week left unclaimed for this lock).
2. Time passes, and during this period, the total voting power (`veSupply`) drops to zero because there are no active locks.
3. The user then creates a new lock and merges it to the expired (`_to` tokenId.) The `merge` function updates the end time of the expired lock to that of the new lock (it's greated then `locked[from].end` which is expired), effectively reactivating the expired one:

```
function merge(uint256 _from, uint256 _to) external override {
    require(_from != _to, "same nft");
    require(_isApprovedOrOwner(msg.sender, _from), "from not approved");
    require(_isApprovedOrOwner(msg.sender, _to), "to not approved");

    LockedBalance memory _locked0 = locked[_from];
    LockedBalance memory _locked1 = locked[_to];
    uint256 value0 = uint256(int256(_locked0.amount));
    uint256 end = _locked0.end >= _locked1.end ? _locked0.end : _locked1.end;
    // more code ...
}
```

4. Notice that, between the time of **expiredLock end time and merging new lock** was a zero `veSupply`, which get recoreded in the `feeDistributer` contract:

```
function _checkpointTotalSupply(uint256 timestamp) internal {
    // prev code ...
    locker.checkpoint();

    for (uint256 index = 0; index < 20; index++) {
        // some code ...
        veSupply[t] = Math.max(uint128(pt.bias - pt.slope * dt), 0);  @audit this will be zero when ever not
↪   token locked in locker contract (ignore the unsafe casting for now )
        // more code ...
    }
    // more code ...
}
```

5. Now when the user tries to claim rewards through the `FeeDistributor` contract, the `claim` function starts the reward calculation from the last unclaimed epoch of the expired lock. The calculation encounters the period where `veSupply` was zero, and since the rewards are proportional to the user's share of `veSupply`, the function attempts to divide by zero, causing a revert.
6. As a result, the user is unable to claim any rewards, and all rewards accumulated for the merged tokenId become unclaimable.

   Also, allowing the `merge` of expired locks permits a scenario where a user can effectively exceed the maximum lock duration. For instance, a user with one NFT locked for the maximum period could, after some time, lock a new NFT with the maximum period also and then merge it into the first, in this case the result is a lock with period from `start` to `end` more then maximum period. However, this does not grant additional voting power or rewards, as the time already elapsed is not restored.

Proof of concept: Here a proof of concept shows the first sceario:

```
contract setup is Test {
    uint constant supply = 1000000000 ether;
    BonusPool bonus;
    VestedZeroLend vestToken;
    VestedZeroLend zeroToken;
    ZeroLocker locker;
    FeeDistributor feeDistributor;
    StakingEmissions emissions;
    StreamedVesting vest;

    function setUp() public virtual {
    // deploy zero token :
    zeroToken = new VestedZeroLend();
    // deploy vest token :
    vestToken = new VestedZeroLend();
    // deploy locker :
    locker = new ZeroLocker();
    locker.initialize(address(zeroToken));
    // deploy feedistributer :
    feeDistributor = new FeeDistributor();
    feeDistributor.initialize(address(locker), address(vestToken));
    // deploy emissions :
    emissions = new StakingEmissions();
    emissions.initialize(feeDistributor, vestToken, 100_000 ether);
    // deploy vest :
    vest = new StreamedVesting();
    // deploy bonus pool :
    bonus = new BonusPool(zeroToken, address(vest));
    vest.initialize(zeroToken, IERC20Burnable(address(vestToken)), locker, bonus);
    // initial balances :
    zeroToken.transfer(address(bonus), 5 * supply);
    zeroToken.transfer(address(vest), 55 * supply);
    vestToken.transfer(address(emissions), 50 * supply);
    vestToken.addwhitelist(address(emissions), true);
    vestToken.addwhitelist(address(feeDistributor), true);
    // disable whitelist from the zero token:
    zeroToken.toggleWhitelist(false, false);
    // start vesting and emissions :
    skip(3 days);
    vest.start();
    emissions.start();
    }

    function skipAndDistribute() internal {
```

```
    skip(1 weeks);
    emissions.distribute();
    }
    function test_pocMergeExpired() public {
    // bob create a lock :
    zeroToken.transfer(bob, 1000 ether);
    vm.startPrank(bob);
    zeroToken.approve(address(locker), 1000 ether);
    uint tokenId1 = locker.createLock(10 ether, 3 weeks);
    for (uint a; a < 5; a++) {
        skipAndDistribute();
    } // skip 5 weeks
    // after first lock expired by 2 weeks , bob create another lock:
    uint tokenId2 = locker.createLock(10 ether, 10 weeks);
    // bob merge the new lock , to the expired lock:
    locker.merge(tokenId2, tokenId1);
    // bob tries to claim rewards, but it will revert due to veSupply is zero in weeks 4 ,5:
    vm.expectRevert();
    feeDistributor.claim(tokenId1);
    }
}
```

Impact: Allowing the merging of expired locks can lead to unexpected behavior within the system.potentially disrupting the intended logic of reward distribution and preventing users from receiving their legitimate Rewards in some scenarios as described above.

Recommendation: Assert that the `tokenId`s to be merged are not expired:

```
  function merge(uint256 _from, uint256 _to) external override {
      require(_from != _to, "same nft");
      require(_isApprovedOrOwner(msg.sender, _from), "from not approved");
      require(_isApprovedOrOwner(msg.sender, _to), "to not approved");

      LockedBalance memory _locked0 = locked[_from];
      LockedBalance memory _locked1 = locked[_to];
+     if (_locked0.end <= block.timestamp || locked1.end <= block.timestamp) revert("expired locks can't be
↪  merged");
      // more code ...
  }
```

3.2.16 `_burn()` approved users cannot execute normally

Submitted by bin2chen, also found by 8olidity, AuditorPraise, cccz, osmanozdemir1, ravikiranweb3, Chinmay Farkya, jovi.eth, atarpara, saucecri, elhaj, Naveen Kumar J - 1nc0gn170, Ch301, pauleth, 0x175, wafflemakr and 0xarno

Severity: Medium Risk

Context: ZeroLocker.sol#L1118-L1120

Description: In `ZeroLocker.sol`, we can grant other users permission to operate `NFT` through `approve()` or `setApprovalForAll()`.

However, the current implementation of `_burn()` is incorrect, resulting in only the `owner` being able to execute `withdraw()` and `merge()`.

```
function _burn(uint256 _tokenId) internal {
    require(
        _isApprovedOrOwner(msg.sender, _tokenId),
        "caller is not owner nor approved"
    );

    address owner = _ownerOf(_tokenId);

    // Clear approval
    _approve(address(0), _tokenId);
    // Remove token
    _removeTokenFrom(msg.sender, _tokenId);
    emit Transfer(owner, address(0), _tokenId);
}
```

There are two problems with the above implementation:

1. The `_approve(address(0), _tokenId)` method internally checks whether `msg.sender` is `owner` or `senderIsApprovedForAll`, and does not allow `idToApprovals[id]`. So even if the user has been granted permission through `approve(user,id)`, they still cannot execute `_burn()`. The correct usage should be: `_clearApproval(owner,_tokenId)`.
2. The `_removeTokenFrom(msg.sender, _tokenId);` method passes in the first parameter as _from = `msg.sender`. The method internally checks `assert(idToOwner[_tokenId] == _from);`. So even if the user be checked by `_isApprovedOrOwner()`, they still cannot execute `_burn()`. The correct usage should be: `_removeTokenFrom(owner, _tokenId)`.

Impact: Even if the user be approved, they still cannot execute `withdraw()` and `merge()`.

Recommendation:

```
  function _burn(uint256 _tokenId) internal {
      require(
          _isApprovedOrOwner(msg.sender, _tokenId),
          "caller is not owner nor approved"
      );

      address owner = _ownerOf(_tokenId);

      // Clear approval
-     _approve(address(0), _tokenId);
+     _clearApproval(owner,_tokenId);
      // Remove token
-     _removeTokenFrom(msg.sender, _tokenId);
+     _removeTokenFrom(owner, _tokenId);
      emit Transfer(owner, address(0), _tokenId);
  }
```

3.2.17 Inequitable reward distribution in `zlrewardscontroller` due to dynamic available rewards adjustment

Submitted by elhaj

Severity: Medium Risk

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

Description: The `ZLRewardsController` contract orchestrates the distribution of rewards to pools based on their allocation points. The intended design is to calculate a total `rawReward` for each update cycle using the `rewardsPerSecond` rate and the time elapsed since the last update. This total is then proportionally divided among pools, with each pool's share determined by its allocation points relative to the `totalAllocPoint`. The goal is to ensure a consistent and fair distribution of rewards across all pools.

The `massUpdatePools` function in the `ZLRewardsController` contract updates each pool's rewards by iterating over them and adjusting the `availableRewards` after each calculation. The rewards for each pool are determined based on the pool's allocation points and the global `rewardsPerSecond` rate.

```
function _massUpdatePools() internal {
    uint256 totalAP = totalAllocPoint;
    uint256 length = poolLength();
    for (uint256 i; i < length; ) {
        _updatePool(poolInfo[registeredTokens[i]], totalAP);
        unchecked {
        i++;
        }
    }
    lastAllPoolUpdate = block.timestamp;
}
function _updatePool(PoolInfo storage pool, uint256 _totalAllocPoint) internal {
    // some code ....
    (uint256 reward, uint256 newAccRewardPerShare) = _newRewards(pool, _totalAllocPoint);
    accountedRewards = accountedRewards + reward;
    pool.accRewardPerShare = pool.accRewardPerShare + newAccRewardPerShare;
    pool.lastRewardTime = timestamp;
}
```

The contract checks if the calculated `rawReward` (based on `rewardsPerSecond` and the elapsed time) is

greater than the `availableRewards`. If so, the `rawReward` is the `rawReward` is adjusted to match the `availableRewards`.

```
function _newRewards(PoolInfo memory pool, uint256 _totalAllocPoint) internal view returns (uint256 newReward,
↪  uint256 newAccRewardPerShare) {
    uint256 lpSupply = pool.totalSupply;
    if (lpSupply > 0) {
        uint256 duration = block.timestamp - pool.lastRewardTime;
        uint256 rawReward = duration * rewardsPerSecond;

        uint256 rewards = availableRewards();
        if (rewards < rawReward) {
            rawReward = rewards;
        }
        newReward = (rawReward * pool.allocPoint) / _totalAllocPoint;
        newAccRewardPerShare = (newReward * ACC_REWARD_PRECISION) / lpSupply;
    }
}
```

This adjustment can occur at any point—whether at the beginning, middle of the pool update process—due to the `availableRewards` decreasing with each pool's reward distribution:

```
function _updatePool(PoolInfo storage pool, uint256 _totalAllocPoint) internal {
    // some code ....
    (uint256 reward, uint256 newAccRewardPerShare) = _newRewards(pool, _totalAllocPoint);
    accountedRewards = accountedRewards + reward;
    // more code ..
}

function availableRewards() internal view returns (uint256 amount) {
    return depositedRewards - accountedRewards;
}
```

When the contract adjusts the `rawReward` to the lower `availableRewards`,there is a logical flaw when distributing rewards that leads to an unfair allocation among pools, which is not the intended behavior. To summarize:

- The contract is designed to distribute `rawReward` based on the `rewardsPerSecond` and the elapsed time since the last update.
- If `availableRewards` is less than `rawReward`, the contract reduces `rawReward` to match `availableRewards`.
- As each pool is updated, `availableRewards` decreases, which can cause subsequent pools to receive less than their fair share based on allocation points.

Here a simplified example: assume we have 3 reward pools, pool1 have `100 AllocPoints`, pool2 have `100 allocPoints`, pool3 have `100 allocPoints`, so the `totalAllocPoints = 300`.

- **Initial Setup:**
  - Pool1 supply: `1000` LP tokens.
  - Pool2 supply: `1000` LP tokens.
  - Pool3 supply: `1000` LP tokens.
  - Rewards per second (`rewardsPerSecond`): 3.
  - Duration since last reward time: `150` seconds.
  - Available rewards (`rewards`): `300` tokens.
  - Raw rewards (calculated as `duration * rewardsPerSecond`):`150 * 3 =``450` tokens. which is greater then availableRewards (`reward` = 300), so we will distribute rewards depending on `availableRewards`.
- **Pool1 Reward Calculation:**
  - New rewards for the first pool: `300 * 100 / 300 = 100` tokens.
  - Remaining available rewards (`rewards`): `200` tokens.
- **Pool2 Reward Calculation:**

  - New rewards for the second pool: `200 * 100 / 300 = 66.6` tokens.
  - Remaining available rewards: `134.4` tokens.
- **Pool3 Reward Calculation:**
  - New rewards for the third pool: `134.4 * 100 / 300 = 44.8` tokens.
  - Remaining rewards after all pools: `89.6` tokens left undistributed.
- This example demonstrates that the later pools receive fewer and fewer rewards than their allocation points warrant,also there is a remainder of rewards that should have been distributed.

Impact: When `rawRewards` are higher than `availableRewards`, the system's design to reduce `rawRewards` leads to later pools receiving a smaller portion of rewards. This is exacerbated as `availableRewards` shrink with each pool update, causing a compounding effect where each subsequent pool gets progressively less, resulting in an inequitable distribution of rewards.

Recommendation: The solution is to maintain the `availableRewards` constant during the distribution to all pools and only subtract the distributed amount (`accountedRewards + reward`) at the end. This ensures that rewards are distributed fairly and proportionally among pools.

It should be somthing like that:

```
  function _massUpdatePools() internal {
  uint256 totalAP = totalAllocPoint;
  uint256 length = poolLength();
+ uint rewards;
  for (uint256 i; i < length; ) {
- _updatePool(poolInfo[registeredTokens[i]], totalAP);
+ rewards += _updatePool(poolInfo[registeredTokens[i]], totalAP);

    unchecked {
        i++;
    }
  }

  lastAllPoolUpdate = block.timestamp;
+ accountedRewards = accountedRewards + reward;
  }

+ function _updatePool(PoolInfo storage pool, uint256 _totalAllocPoint) internal returns (uint) {
  uint256 timestamp = block.timestamp;
  uint256 endReward = endRewardTime();
  if (endReward <= timestamp) {
      timestamp = endReward;
  }
  if (timestamp <= pool.lastRewardTime) {
+     return 0;
  }

  (uint256 reward, uint256 newAccRewardPerShare) = _newRewards(pool, _totalAllocPoint);
- accountedRewards = accountedRewards + reward;
  pool.accRewardPerShare = pool.accRewardPerShare + newAccRewardPerShare;
  pool.lastRewardTime = timestamp;
+ return reward; //@remind : this is not the case , cause the rewards calculated till now,and timestamp can be
↪  endReward?
  }

  function claim(address _user, address[] memory _tokens) public whenNotPaused {
      // prev code ...

-     _updatePool(pool, _totalAllocPoint);
+     accountedRewards_+= updatePool(pool, _totalAllocPoint);

      // more code ...
  }

  function _handleActionAfterForToken(address _token, address _user, uint256 _balance, uint256 _totalSupply)
↪  internal {

      // prev code ...

-     _updatePool(pool, totalAllocPoint);
+     accountedRewards_+= updatePool(pool, _totalAllocPoint);
```

```
    // more code ...
}
```