SHERLOCK

Security Review For Beam

Collaborative Audit Prepared For: **Beam**
Lead Security Expert(s): **AlexMurphy**
TessKimy
Date Audited: **September 11 - October 30, 2025**

Introduction

Smart contracts for a ve(3,3) protocol. Provides the following features: DEX token emissions, locking as veNFT positions, rebase distribution, epoch based voting, rewards distribution, volatile/stable pairs DEX, and Algebra farming integration for concentrated liquidity positions. Forked from Thena / Retro. Initially developed for Beam DEX.

Scope

Repository: Celeborn2BeAlive/beam-ve33-contracts

Audited Commit: e9a2d73e031f607098c8e2dcee470ea362473e75

Final Commit: cd237400926d9f8e4e150cf8572a750176e01b87

Files:

- contracts/algebra/AlgebraVaultFactory.sol
- contracts/algebra/AlgebraVault.sol
- contracts/algebra/GaugeEternalFarming.sol
- contracts/algebra/IncentiveMakerUpgradeable.sol
- contracts/algebra/interfaces/IAlgebraVaultFactory.sol
- contracts/algebra/interfaces/IAlgebraVault.sol
- contracts/algebra/interfaces/IIncentiveMaker.sol
- contracts/alms/ALMFeeVault.sol
- contracts/Claimer.sol
- contracts/EmissionToken.sol
- contracts/EpochDistributorUpgradeable.sol
- contracts/GaugeFactory.sol
- contracts/Gauge.sol
- contracts/GlobalFactory.sol
- contracts/interfaces/IEmissionToken.sol
- contracts/interfaces/IEpochDistributor.sol
- contracts/interfaces/IERC20.sol
- contracts/interfaces/IFeeVault.sol
- contracts/interfaces/IGaugeFactory.sol
- contracts/interfaces/IGauge.sol

- contracts/interfaces/IGlobalFactory.sol
- contracts/interfaces/IMinter.sol
- contracts/interfaces/IPairFactory.sol
- contracts/interfaces/IPairInfo.sol
- contracts/interfaces/IRebaseDistributor.sol
- contracts/interfaces/IVeArtProxy.sol
- contracts/interfaces/IVoter.sol
- contracts/interfaces/IVotingEscrow.sol
- contracts/interfaces/IVotingIncentivesFactory.sol
- contracts/interfaces/IVotingIncentives.sol
- contracts/interfaces/IWeightedPoolsSimple.sol
- contracts/libraries/Base64.sol
- contracts/libraries/GaugeMath.sol
- contracts/libraries/Math.sol
- contracts/MinterUpgradeable.sol
- contracts/Proxies.sol
- contracts/RebaseDistributor.sol
- contracts/solidly/interfaces/IDibs.sol
- contracts/solidly/interfaces/IPairCallee.sol
- contracts/solidly/interfaces/IPair.sol
- contracts/solidly/PairFactoryUpgradeable.sol
- contracts/solidly/PairFees.sol
- contracts/solidly/Pair.sol
- contracts/solidly/RouterV2.sol
- contracts/test/ERC20.sol
- contracts/test/TestAlgebraEternalFarming.sol
- contracts/test/TestAlgebraFactory.sol
- contracts/test/TestAlgebraPool.sol
- contracts/VeArtProxy.sol
- contracts/Voter.sol
- contracts/VotingEscrowERC20.sol

- contracts/VotingEscrow.sol
- contracts/VotingIncentivesFactory.sol
- contracts/VotingIncentives.sol

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
2	1	2

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: [ve(3,3] Attacker can steal new emissions from Rebase Distributor [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-beam-sept-11th/issues/125

Summary

Attacker can steal new emissions due to block number based accounting in Rebase Distributor. `block.number` is not a good way to track weekly total locking in voting escrow and due to usage of this method attacker can steal rebase tokens.

Vulnerability Detail

When we reach a new week, `update_period` is called from Minter contract. It firstly calls `checkpoint_token` and then it calls `checkpoint_total_supply` function.

1. `checkpoint_token` tracks current block number and distributed amount
2. `_checkpoint_total_supply` function calculates the total voting power of entire community

While claiming the rewards following method is used:

```
uint id_bal = IVotingEscrow(voting_escrow).balanceOfAtNFT(id, time_to_block[t]);
uint share =  id_bal * 1e18 / ve_supply[t];
```

This is not a safe way to calculate his balance. Attacker can update the period and then deposits his own lock with very big amount in the same block. `_checkpoint_total_supply` won't account this deposit but `balanceOfAtNFT` will account it.

Impact

Loss of fund on emission tokens. Attacker can also steal from previous emissions if following expression is bigger 1.0 ratio:

`uint share =  id_bal * 1e18 / ve_supply[t];`

Tool Used

Manual Review

Recommendation

Following fix solves the problem:

```
    function _checkpoint_total_supply() internal {
        address ve = voting_escrow;
        uint t = time_cursor;
        uint rounded_timestamp = block.timestamp / WEEK * WEEK;
        IVotingEscrow(ve).checkpoint();

        for (uint i = 0; i < 20; i++) {
            if (t > rounded_timestamp) {
                break;
            } else {
                uint epoch = _find_timestamp_epoch(ve, t-1);
                IVotingEscrow.Point memory pt =
                 ↪  IVotingEscrow(ve).point_history(epoch);
                int128 dt = 0;
                if (t - 1 > pt.ts) {
                    dt = int128(int256(t - 1 - pt.ts));
                }
                ve_supply[t] = Math.max(uint(int256(pt.bias - pt.slope * dt)), 0);
            }
            t += WEEK;
        }

        time_cursor = t;
    }

    function _find_user_timestamp_point(address ve, uint _timestamp, uint tokenId)
     ↪  internal view returns (IVotingEscrow.Point memory) {
        uint _min = 0;
        uint _max = IVotingEscrow(ve).user_point_epoch(tokenId);
        for (uint i = 0; i < 128; i++) {
            if (_min >= _max) break;
            uint _mid = (_min + _max + 2) / 2;
            IVotingEscrow.Point memory pt =
             ↪  IVotingEscrow(ve).user_point_history(tokenId, _mid);
            if (pt.ts <= _timestamp) {
                _min = _mid;
            } else {
                _max = _mid - 1;
            }
        }
        return IVotingEscrow(ve).user_point_history(tokenId, _min);
    }

...

    function _toClaim(uint id, uint t) internal view returns(uint to_claim) {

        IVotingEscrow.Point memory userData =
         ↪  IVotingEscrow(voting_escrow).user_point_history(id,1);
        if(ve_supply[t] == 0) return 0;
        if(tokens_per_week[t] == 0) return 0;
        if(userData.ts > t) return 0;

        //uint id_bal = IVotingEscrow(voting_escrow).balanceOfNFTAt(id, t);
        //uint id_bal = IVotingEscrow(voting_escrow).balanceOfAtNFT(id,
        ↪  time_to_block[t]);
        IVotingEscrow.Point memory pt = _find_user_timestamp_point(voting_escrow,
        ↪  t-1, id);
        int128 dt = 0;
        if (t - 1 > pt.ts) {
            dt = int128(int256(t - 1 - pt.ts));
        }
        uint id_bal = Math.max(uint(int256(pt.bias - pt.slope * dt)), 0);
        uint share = id_bal * 1e18 / ve_supply[t];

        to_claim = share * tokens_per_week[t] / 1e18;
    }

...
```

Issue H-2: [ve(3,3] Incorrect variable accounting for setting rates in IncentiveMaker [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-beam-sept-11th/issues/128

Summary

Incentive maker contract is responsible to add rewards for algebra pools and update the distribution rate for these pools. There are 2 tokens handled in IncentiveMaker contract: emission token and wGasToken. However, while updates it sets the other tokens distribution rate to zero which locks token in the farming contract.

Vulnerability Detail

In incentive maker contract following functions are used for adding reward and setting reward rate:

1. `updateIncentive`
2. `updateExtraIncentive`

`updateIncentive` is called while emission token distribution through gauges. It transfers the tokens and then sets the distribution rate to `reward/week` for emission token and 0 for wGasToken. If wGasToken is still distributed while this call, remaining tokens will be stuck in AlgebraEternalFarming contract. Same scenario is exist for `updateExtraIncentive` function. It sets the rate of emission token distribution to 0.

Currently, distributing these two tokens in the same time is not possible due to incorrect `setRates` call.

Impact

Emission token/wGasToken will be stuck in AlgebraEternalFarming, it will cause loss of funds for gauge depositors.

Code Snippet

https://github.com/sherlock-audit/2025-09-beam-sept-11th/blob/a0987b4f1ae536301178a99d9c68e8f4aaeaca00/beam-ve33-contracts/contracts/algebra/IncentiveMakerUpgradeable.sol#L196-L198

https://github.com/sherlock-audit/2025-09-beam-sept-11th/blob/a0987b4f1ae536301178a99d9c68e8f4aaeaca00/beam-ve33-contracts/contracts/algebra/IncentiveMakerUpgradeable.sol#L221-L223

Tool Used

Manual Review

Recommendation

> Note: It should also account leftover tokens from previous distribution. If reward is added before previous one ends it will set a lower distribution rate than actual.

It should check the current rate for other tokens distribution and set it in order to keep it as it is.

Discussion

Celeborn2BeAlive

For this issue I chose to remove the support for extra incentives instead of tracking their reward rate to re-set it properly. Tracking them requires to interact with virtual pools managed by Algebra, and as such more integration testing in a forked environment which could require much more time and additional audit work.

We won't use the feature anyway, and if we change our mind we'll allocate time to re-implement the IncentiveMaker properly, do the testing, an additional audit, and an upgrade of the smart contract for deployment.

About the note I agree, but similarly fixing that would require more dev time & testing. Also given the epoch-based execution flow of the protocol, all past rewards should have been distributed before a new distribution occurs.

Do you agree with these choices or should I re-consider ?

Issue M-1: [ve(3,3)] Owner will be unable to remove reward tokens from the Gauge contract [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-beam-sept-11th/issues/122

Summary

An inverted boolean check in `removeRewardToken()` will cause the function to always revert when attempting to remove valid reward tokens, making it impossible for the owner to properly manage the reward token list as the function can only be called on tokens that are NOT reward tokens.

Vulnerability Detail

The function `removeRewardToken()` is intended to remove existing reward tokens from the `rewardTokens` array and update the `isRewardToken` mapping. However, the require statement at line 155 checks that the token is NOT a reward token (`!isRewardToken[_token]`), when it should verify that the token IS a reward token before attempting removal.

Impact

The owner cannot remove reward tokens from the gauge contract, breaking the intended administrative functionality for managing reward tokens.

Code Snippet

https://github.com/sherlock-audit/2025-09-beam-sept-11th/blob/a0987b4f1ae536301178a99d9c68e8f4aaeaca00/beam-ve33-contracts/contracts/Gauge.sol#L154

Recommendation

The correct check should be `require(isRewardToken[_token])`.

Discussion

Celeborn2BeAlive

Fixed: https://github.com/Celeborn2BeAlive/beam-ve33-contracts/commit/eb9ef5aa4a0b61b12a8ce673666fcc9c5c550e4a

Issue L-1: [ve(3,3)] Repeated `merge()` operations will inflate the `supply` accounting variable beyond actual locked tokens [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-beam-sept-11th/issues/127

Vulnerability Detail

The VotingEscrow contract tracks the total amount of emission tokens locked across all veNFT positions in the `supply` state variable. When users lock tokens, the `supply` increases; when they withdraw, it decreases. Operations like `split()` that redistribute locked tokens without changing the total amount correctly maintain supply neutrality by first decreasing supply before calling `_deposit_for()`, which increases it again.

However, the `merge()` function contains an accounting inconsistency. The function allows a user to merge two veNFT positions (`_from` and `_to`) into a single position. This operation should be supply-neutral since no new tokens enter or leave the system - tokens are merely consolidated.

The issue occurs because the supply variable is increased in the `_deposit_for()` function. However, unlike `withdraw()` which decreases supply when removing locked tokens, or `split()` which explicitly decreases supply before redistributing, the `merge()` function never decreases supply when clearing the _from position. The net result is that each `merge()` operation inflates the supply variable by the amount locked in the source token, even though the actual total locked tokens remain unchanged.

The following test can be used to confirm the behavior:

```
function test_merge_double_count_supply() public {
        // Create lock for Alice and Bob
        _createLock(alice, 100e18, 365 days);
        _createLock(alice, 100e18, 365 days);

        assertEq(votingEscrow.supply(), 200e18);

        vm.prank(alice);
        votingEscrow.merge(1, 2);

        assertEq(votingEscrow.supply(), 300e18);
}
```

Impact

The protocol suffers from corruption of accounting data that may be relied upon by external integrations or protocol logic that reads the `supply` variable.

Code Snippet

https://github.com/sherlock-audit/2025-09-beam-sept-11th/blob/a0987b4f1ae536301178a99d9c68e8f4aaeaca00/beam-ve33-contracts/contracts/VotingEscrow.sol#L1063-L1078

Recommendation

Decrease the `supply` variable before burning the source token in the `merge()` function, similar to how it's handled in `split()`.

Issue L-2: [ve(3,3)] Precision loss causes less reward distribution in Gauge [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-beam-sept-11th/issues/129

Summary

Gauge is responsible for distribution of emission tokens and other extra rewards published by managers. In `notifyRewardAmount` function `_rewardRate` is calculated without adding an extra precision.

Vulnerability Detail

`_rewardRate` calculated with following formula:

```
        if (block.timestamp >= _periodEnd) {
@>          _rewardRate[_token] = _amount / DURATION;
        } else {
            uint256 _rewRate = _rewardRate[_token];
            uint256 leftover = (_periodEnd - block.timestamp) * _rewRate;
@>          _rewardRate[_token] = (_amount + leftover) / DURATION;
        }
```

Rate calculation needs extra precision in order to keep distribution accuracy high, however in here we don't add extra precision to amount and DURATION is equal to 604,800. It makes reward rate inaccurate due to precision loss.

Impact

Users will get less reward than expected due to precision loss. This loss is not significant for tokens which has 18 decimals and emission token won't be affected significantly but it will be significant for WBTC.

Code Snippet

https://github.com/sherlock-audit/2025-09-beam-sept-11th/blob/a0987b4f1ae536301178a99d9c68e8f4aaeaca00/beam-ve33-contracts/contracts/Gauge.sol#L445-L452

Tool Used

Manual Review

Recommendation

Consider applying following changes for accurate reward distribution:

```
        if (block.timestamp >= _periodEnd) {
-           _rewardRate[_token] = _amount / DURATION;
+           _rewardRate[_token] = _amount * 1e18 / DURATION;
        } else {
            uint256 _rewRate = _rewardRate[_token];
            uint256 leftover = (_periodEnd - block.timestamp) * _rewRate;
-           _rewardRate[_token] = (_amount + leftover) / DURATION;
+           _rewardRate[_token] = (_amount * 1e18 + leftover ) / DURATION;
        }

...

-         require(_rewardRate[_token] <= balance / DURATION, "!rate");
+         require(_rewardRate[_token] <= balance * 1e18 / DURATION, "!rate");

...

    function _earned(address _account, address _token) internal view
     ↪  returns(uint256) {
-       uint256 _newRewards = _balances[_account] * (_rewardPerToken(_token) -
 ↪  _userRewardPerTokenPaid[_account][_token]) / 1e18;
+       uint256 _newRewards = _balances[_account] * (_rewardPerToken(_token) -
 ↪  _userRewardPerTokenPaid[_account][_token]) / 1e36;
        return  _newRewards + _rewards[_account][_token];
    }
```

Disclaimers