SHERLOCK

Security Review For Reppo

Collaborative Audit Prepared For: **Reppo**
Lead Security Expert(s): **hildingr**
newspacexyz
Date Audited: **March 13 - March 18, 2026**

Introduction

Reppo is building the world's first blockchain network of AI Training Data Markets – akin to Scale AI on-chain – where each data market leverages prediction market mechanics to curate AI training datasets for AI labs, robotics, and agentic use cases. Rather than coordinating data curation through off chain contracts, Reppo coordinates data through prediction markets, enabling everyone in the AI training data pipeline to share the upside without intermediaries.

This dual-sided economic commitment directly addresses the AI Training Data Trilemma of quality, scale, and cost. Curated datasets are then monetized permissionlessly on Reppo.exchange, the world's first data DEX. Together, these systems form an open, auditable, and incentive-aligned ecosystem for producing high-signal AI training data at planetary scale.

Scope

Repository: Reppo-Labs/economic-contracts

Audited Commit: b4030e19c8335d9174ec7baaa8801f3cdf9fcecf

Final Commit: 85b050efa954a9c24222a2ce2d79be95569c4482

Files:

- script/DeployV2.s.sol
- src/PodManagerV2.sol
- src/SubnetManager.sol

Final Commit Hash

85b050efa954a9c24222a2ce2d79be95569c4482

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
1	8	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Emission distribution based on total votes incentivizes controversial pods [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/4

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The emission distribution mechanism allocates rewards proportional to total votes (upvotes + downvotes) rather than net positive votes (upvotes - downvotes). This creates a perverse incentive where controversial, barely-passing pods receive significantly more emissions than unanimously-supported high-quality pods with the same upvote count.

Vulnerability Detail

In claimPodOwnerEmissions() and claimVoterEmissions(), pod emission shares are calculated using:

```
uint256 podTotalVotes = podTotalUpVotes + podTotalDownVotes;
uint256 podEmissionsShare = (reppoEmissionsPerEpoch * podTotalVotes) /
↪  subnetTotalVotes;
```

While pods must be net positive (more upvotes than downvotes) to claim rewards, the amount of emissions is based on total vote volume, not quality/net approval.

Concrete Example:

Assume 300 total subnet votes and 300 REPPO emissions per epoch:

Pod A (Controversial):

100 upvotes 99 downvotes Total: 199 votes (net +1, barely passing) Emission share: (300 × 199) / 300 = 199 REPPO Pod B (High Quality):

100 upvotes 0 downvotes Total: 100 votes (net +100, unanimous support) Emission share: (300 × 100) / 300 = 100 REPPO Result: The controversial pod with only 1 net vote receives nearly 2x the emissions of the unanimously-supported pod, despite being objectively lower quality.

Impact

This gives pod owners the opportunity to obtain higher emissions by manipulating podTotalUpVotes. For example, if Pod has 99 upvotes and 100 downvotes, the Pod owner can claim a large emission for 199 with only 2 upvotes.

Code Snippet

https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/PodManagerV2.sol#L201-L203

Tool Used

Manual Review

Recommendation

Allocate emissions based on net positive votes rather than total votes to properly incentivize quality over controversy:

Comparison:

Metric	Pod A (100↑/99↓)	Pod B (100↑/0↓)
Current (Total)	199 votes = 66%	100 votes = 33%
Recommendation (Net)	1 vote = 1%	100 votes = 99%

Discussion

cartmanBruh96

Its a business requirement to collect feedback (either upvotes or downvotes). So the final pod emissions allocation is based on total feedback, hence the above logic. If user feedback is high for a certain pod (regardless of the feedback being either up or down), emissions are allocated proportionally. However the pod owners are penalized if their pod ends up in a net negative situation.

Issue M-1: Subnet owner can retroactively change economic parameters after votes are cast but before claims [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/5

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Subnet owners can modify critical economic parameters (emissions amounts, share percentages, tax rates) after users have voted but before they claim rewards. This breaks the social contract where users vote based on advertised parameters and expect consistent reward calculations, enabling various forms of bait-and-switch attacks.

Vulnerability Detail

When users claim reppo earning or emissions, `SubnetManager.claimReppoEarnings` and `PodManagerV2.claimPodOwnerEmissions()` and `PodManagerV2.claimVoterEmissions()` functions read subnet parameters from current state () rather than the values that existed during the voting epoch.

`seederFeeShare`, podOwnerEmissionSharePercentage`and`reppoEmissionsPerEpoch`can be modified by subnet owners at any time using setter functions like` setSeederFeeShare, setPodOwnerEmissionSharePercentage()`and` setReppoEmissionsPerEpoch()‘.

Same issue appears with `taxRate`.

Attack Scenario:

1. Epoch N: Subnet advertises attractive parameters (10,000 REPPO emissions, 30% owner share)
2. Users vote based on these parameters
3. Epoch N ends, voting tallied
4. Epoch N+1: Before anyone claims, subnet owner changes parameters (100 REPPO emissions, 90% owner share)
5. Users claim and receive drastically reduced rewards

Impact

Subnet owners benefit from manipulating parameters after attracting votes with favorable terms.

Code Snippet

https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/SubnetManager.sol#L394 https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/PodManagerV2.sol#L176 https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/PodManagerV2.sol#L196 https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/PodManagerV2.sol#L200

Tool Used

Manual Review

Recommendation

Implement epoch-based parameter snapshots to ensure immutability of economic rules after voting begins.

Add snapshot storage structure:

```
struct EpochSubnetSnapshot {
    uint256 reppoEmissionsPerEpoch;
    uint256 primaryTokenEmissionsPerEpoch;
    uint8 podOwnerEmissionSharePercentage;
    uint8 taxRate;
    uint8 seederFeeShare;
    bool initialized;
}

// In SubnetStorage
mapping(uint256 subnetId => mapping(uint256 epoch => EpochSubnetSnapshot)) public
 ↪  epochSnapshots;
```

Discussion

cartmanBruh96

Business is aware of this situation and this behaviour is accepted. In the current context, epochs are just 2 days, if the subnet owner shows such kind of behaviour (changing subnet params in bad faith), users will quickly recognise this malicious activities, and refrain from participating in the subnet further. In the application layer, subnets parameters will be made public, with the change history, so subnet participants can make informed decisions regarding their activities within the subnet.

Issue M-2: Tie-Vote griefing attack locks pod emissions [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/6

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The emission claim logic blocks both pod owners and voters from claiming rewards when `podTotalUpVotes == podTotalDownVotes`. This condition enables a **griefing attack** where an attacker (or a malicious subnet owner) intentionally forces a vote tie to permanently prevent reward distribution.

Vulnerability Detail

The emission claiming logic has asymmetric conditions:

In `claimPodOwnerEmissions()` (line 192-193):

```
if (podTotalDownVotes >= podTotalUpVotes) {
    revert PodHasMoreDownVotesThanUpVotes();
}
```

This reverts when downvotes upvotes (includes net-negative AND net-neutral).

In `claimVoterEmissions()` (line 273-274):

```
if (podTotalUpVotes == podTotalDownVotes) {
    revert PodNetNeutralNoEmissions();
}
```

This explicitly reverts for the net-neutral case.

An attacker (or a malicious subnet owner) can cast a **single strategic downvote** on a pod that would otherwise distribute emissions, forcing the vote count into a tie. Once the tie condition is reached, the contract prevents all claims, causing the pod's allocated emissions to become permanently locked in the contract's seeding balance.

Impact

This allows malicious subnet owner to **intentionally sabotage reward distribution**, effectively capturing or freezing emissions that should have been distributed to pod owners and voters.

Code Snippet

https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/PodManagerV2.sol#L192-L194

https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/PodManagerV2.sol#L273-L275

Tool Used

Manual Review

Recommendation

For emissions that pod owners or voters cannot claim, consider redirecting them to a third party (e.g., the `governanceReserve`) or rolling them over into the next epoch's emissions.

Discussion

cartmanBruh96

In case of a tie, emissions will cumulate within the subnet, and will be available up for grabs in the coming epochs.

Issue M-3: Seeded REPPO and primary tokens could be permanently locked in `PodManagerV2` contract [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/7

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `PodManagerV2` contract allows subnet owners and external users to seed REPPO and primary tokens into subnet emission pools through `seedREPPOEmissionsBySubnetOwner()`, `seedPrimaryTokenEmissionsBySubnetOwner()`, and `seedREPPOEmissionsExternal()` functions. However, the contract lacks any mechanism to withdraw unspent seeded tokens. If a subnet becomes inactive or the ecosystem shuts down before all seeded tokens are distributed through emissions, these tokens will be permanently locked in the contract with no way to recover them.

Impact

permanently locked of tokens

Code Snippet

.

Tool Used

Manual Review

Recommendation

Implement a withdrawal mechanism to handle unspent seeded tokens.

```
/// @notice Allows admin to withdraw tokens in emergency situations
/// @param token The ERC20 token to withdraw
/// @param to The address to send tokens to
/// @param amount The amount to withdraw
function withdrawToken(
    IERC20 token,
    address to,
    uint256 amount
) external onlyRole(DEFAULT_ADMIN_ROLE) {
```

```
    token.safeTransfer(to, amount);
    emit TokenWithdrawn(address(token), to, amount);
}
```

Discussion

cartmanBruh96

Business has decided not to implement an withdraw function which might indicate to the community of a potential rug pull opportunity.

Issue M-4: REPPO Seeding Functions Bypass Governance Tax [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/11

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`seedREPPOEmissionsBySubnetOwner()` and `seedREPPOEmissionsExternal()` in `PodManagerV2` do not deduct a governance tax before crediting the seeding balance, while the equivalent primary token function `seedPrimaryTokenEmissionsBySubnetOwner()` does. This results in lost tax revenue to the governance reserve and creates a tax arbitrage favouring REPPO-denominated seedings.

Vulnerability Detail

When a subnet owner seeds emissions using a primary token, `seedPrimaryTokenEmissionsBySubnetOwner()` computes a tax based on `getTaxRate()`, transfers it to the `governanceReserve`, and only credits the post-tax amount:

PodManagerV2.sol#L537-L559

```
function seedPrimaryTokenEmissionsBySubnetOwner(uint256 subnetId, uint256 amount)
↪   public {
    PodManagerStorage storage $ = _getPodManagerStorage();
    // ...
    address primaryTokenAddress = $.subnetManager.getSubnetPrimaryToken(subnetId);
    uint256 balanceBefore = IERC20(primaryTokenAddress).balanceOf(address(this));
    IERC20(primaryTokenAddress).safeTransferFrom(msg.sender, address(this), amount);
    uint256 balanceAfter = IERC20(primaryTokenAddress).balanceOf(address(this));
    uint256 actualAmountReceived = balanceAfter - balanceBefore;
    uint256 tax = (actualAmountReceived * $.subnetManager.getTaxRate()) / 100;
    if (tax > 0) {
        IERC20(primaryTokenAddress).safeTransfer($.governanceReserve, tax);
    }
    uint256 amountAfterTax = actualAmountReceived - tax;
    $.subnetPrimaryTokenSeedingBalance[subnetId] += amountAfterTax;
    emit PrimaryTokenEmissionsSeededBySubnetOwner(subnetId, amountAfterTax,
    ↪   msg.sender);
}
```

However, both REPPO seeding functions skip the tax entirely and credit the full transferred amount:

PodManagerV2.sol#L522-L535

```
function seedREPPOEmissionsBySubnetOwner(uint256 subnetId, uint256 amount) public {
    PodManagerStorage storage $ = _getPodManagerStorage();
    // ...
    $.reppo.safeTransferFrom(msg.sender, address(this), amount);
    $.subnetREPPOSeedingBalance[subnetId] += amount; // @audit full amount, no tax
    emit ReppoEmissionsSeededBySubnetOwner(subnetId, amount, msg.sender);
}
```

PodManagerV2.sol#L562-L583

```
function seedREPPOEmissionsExternal(uint256 subnetId, uint256 amount) public {
    PodManagerStorage storage $ = _getPodManagerStorage();
    // ...
    $.reppo.safeTransferFrom(msg.sender, address(this), amount);
    $.subnetREPPOSeedingBalance[subnetId] += amount; // @audit full amount, no tax
    // ...
    emit ReppoEmissionsSeededExternally(subnetId, amount, msg.sender);
}
```

Impact

Lost governance revenue: Every REPPO-denominated seeding bypasses the tax, depriving the governance reserve of its intended share. Over time this compounds as REPPO seedings accumulate across all subnets.

Recommendation

Apply the same tax logic used in `seedPrimaryTokenEmissionsBySubnetOwner()` to both REPPO seeding functions.

Discussion

cartmanBruh96

Reppo seedings are exempted from tax to encourage seeding with REEPO instead of other tokens

Issue M-5: Minting and Republishing Fees Bypass the Earnings Pool, Denying External Seeders Their Share [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/12

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Publishing and republishing fees in `PodManagerV2` are sent directly to the `subnetOwner` instead of being routed through the `SubnetManager` earnings pool. External seeders, who fund emissions to incentivize content on a subnet, receive no portion of these fees despite the protocol defining a `seederFeeShare` split on subnet earnings.

Vulnerability Detail

External seeders deposit REPPO (or primary tokens) into a subnet's emission pool via `seedREPPOEmissionsExternal()`. These funds pay for pod owner and voter rewards each epoch. In return, seeders are entitled to a share of the subnet's earnings, controlled by the `seederFeeShare` parameter (default 50%).

The earnings pool that seeders can claim from is `SubnetManager.earnings[subnetId].epochEarnings[epoch]`. Currently, only subnet access fees feed this pool:

SubnetManager.sol#L300-L323

```
function accessSubnetWithREPPOFee(uint256 subnetId, address to) public {
    // ...
    uint256 netAmount = accessFee - taxAmount;
    $.earnings[subnetId].epochEarnings[currentEpoch].reppoEarnings += netAmount;
    // ...
}
```

However, the four minting and republishing functions in `PodManagerV2` transfer fees directly to the `subnetOwner`, completely bypassing the earnings pool:

PodManagerV2.sol#L359-L385

```
function mintPodWithREPPO(address to, uint256 subnetId) external returns (uint256
↪  podId) {
    // ...
    uint8 taxRate = $.subnetManager.getTaxRate();
    uint256 tax = (publishingFee * taxRate) / 100;
    if (tax > 0) {
        $.reppo.safeTransfer($.governanceReserve, tax);
```

```
    }
    address subnetOwner = $.subnetManager.subnetOwner(subnetId);
    $.reppo.safeTransfer(subnetOwner, publishingFee - tax); // @audit sent directly
    ↪  to subnetOwner
    // ...
}
```

The same pattern is repeated in:

- `mintPodWithPrimaryToken()` (PodManagerV2.sol#L388-L419): sends `actualAmountReceived - tax` directly to `subnetOwner`
- `republishPodWithREPPO()` (PodManagerV2.sol#L450-L481): sends `republishFee - tax` directly to `subnetOwner`
- `republishPodWithPrimaryToken()` (PodManagerV2.sol#L484-L519): sends `actualAmountReceived - tax` directly to `subnetOwner`

Meanwhile, the `claimReppoEarnings()` function in `SubnetManager` is designed to split all epoch earnings between the subnet owner and external seeders:

SubnetManager.sol#L377-L428

```
function claimReppoEarnings(uint256 subnetId, uint256 epoch) public {
    // ...
    uint8 seederFeeShare = $.meta[subnetId].seederFeeShare;
    uint256 seedersShare = (epochEarnings * seederFeeShare) / 100;
    uint256 ownerShare = epochEarnings - seedersShare;
    // ...
    // External seeders claim their pro-rata share of seedersShare
    uint256 amountToClaim = (seedersShare * callerSeedings) / totalSeedings;
    $.reppo.safeTransfer(msg.sender, amountToClaim);
}
```

Publishing and republishing fees represent real protocol revenue generated on the subnet, yet they never enter the `epochEarnings` mapping, so the `seederFeeShare` split is never applied to them. External seeders who fund the emissions that drive content creation and user engagement on the subnet are excluded from this revenue stream entirely.

Impact

Loss of funds for external seeders. All minting and republishing fee revenue (after governance tax) flows exclusively to the `subnetOwner`, bypassing the `seederFeeShare` split that is enforced for access fees.

Recommendation

We could route the post-tax minting and republishing fees through the `SubnetManager` earnings pool so the `seederFeeShare` split applies consistently to all subnet revenue.

Discussion

cartmanBruh96

External seeders will seed based on the future potential and the willingness to see the subnet succeed. Only incentive they are eligible is a portion of the access fees.

Issue M-6: Pod Owners Can Republish at Any Time, Enabling Front-Running of Parameter Changes [RESOLVED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/13

Summary

`republishPodWithREPPO()` and `republishPodWithPrimaryToken()` in `PodManagerV2` impose no requirement that a pod be near expiry before republishing. Combined with the additive validity model, a pod owner can front-run governance tax increases, validity period reductions, or fee increases by calling republish repeatedly to stack validity at the old, cheaper terms.

Vulnerability Detail

Both republish functions use an additive validity model where the configured `podValidityEpochs` is added to the pod's current validity, with no check on remaining time:

PodManagerV2.sol#L450-L481

```
function republishPodWithREPPO(
    uint256 podId
) external onlyPodOwner(podId) {
    PodManagerStorage storage $ = _getPodManagerStorage();
    uint256 subnetId = $.podSubnet[podId];

    uint256 republishFee = $.subnetManager.getRepublishFeeREPPO(subnetId);
    if (republishFee > 0) {
        uint8 taxRate = $.subnetManager.getTaxRate();
        uint256 tax = (republishFee * taxRate) / 100;
        // ... fee transfers ...
    }

    uint256 podValidityEpochs = $.subnetManager.getPodValidityEpochs(subnetId);
    uint256 currentPodValidity = $.podEpochValidity[podId];
    $.podEpochValidity[podId] = currentPodValidity + podValidityEpochs; // @audit
    ↪  additive, no expiry check
    emit PodRepublished(podId, msg.sender);
}
```

The same pattern is used in `republishPodWithPrimaryToken()`:

PodManagerV2.sol#L484-L519

```
function republishPodWithPrimaryToken(
```

```
    uint256 podId
) external onlyPodOwner(podId) {
    // ... fee logic ...
    uint256 currentPodValidity = $.podEpochValidity[podId];
    uint256 podValidityEpochs = $.subnetManager.getPodValidityEpochs(subnetId);
    $.podEpochValidity[podId] = currentPodValidity + podValidityEpochs; // @audit
    ↪  same additive pattern
    emit PodRepublished(podId, msg.sender);
}
```

Several parameters that directly affect the cost and benefit of republishing are changeable by governance or the subnet owner:

- `setTaxRate()` (SubnetManager.sol#L158-L165): governance can increase the tax rate up to 50%
- `setPodValidityEpochs()` (SubnetManager.sol#L220-L227): subnet owner can reduce the validity period granted per republish
- `setRepublishFeeREPPO()` / `setRepublishFeePrimaryToken()` (SubnetManager.sol#L200-L217): subnet owner can increase republish fees

A pod owner who observes any of these parameter-change transactions in the mempool can front-run them by calling republish multiple times in a single batch. For example, if `podValidityEpochs` is currently 4 and the subnet owner submits a transaction to reduce it to 2, the pod owner can call republish 10 times before the reduction takes effect, stacking `10 * 4 = 40` additional epochs of validity at the old rate. After the parameter change, each republish would only grant 2 epochs, so the attacker effectively locked in double the validity per fee paid.

Impact

Pod owners can lock in long validity periods at a discount by front-running parameter changes. This undermines the ability of governance and subnet owners to adjust economic parameters, as existing pod owners can preemptively extend their validity far into the future before any increase in cost or decrease in validity takes effect.

Recommendation

Require the pod to be near expiry (e.g., within 1 epoch of expiration) before allowing republish. Alternatively, replace the additive validity model with an absolute reset that sets validity to `currentEpoch + podValidityEpochs` rather than `currentPodValidity + podValidityEpochs`, preventing validity stacking entirely.

Discussion

cartmanBruh96

1. updated to stack based on latest epoch
2. front running is fine as we are planning the give users the option to pre-publish for future epochs (rather than them having to come back again every 4 epochs and republish) in the application layer

Issue M-7: Insufficient Seed for One Token Type Blocks Emission Claims for Both Tokens [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/14

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`claimPodOwnerEmissions()` and `claimVoterEmissions()` in `PodManagerV2` process both REPPO and primary token emission payouts within a single transaction. If the seeding balance for either token is insufficient, the entire transaction reverts, blocking the claimant from receiving the token that does have sufficient seed. This can indefinitely freeze all emission claims if one token type is never re-seeded.

Vulnerability Detail

Both claim functions read the seeding balances for REPPO and the primary token at the start of execution, then process each token branch sequentially. Each branch reverts if its seed is insufficient:

PodManagerV2.sol#L164-L239

```
function claimPodOwnerEmissions(uint256 podId, uint256 epoch) external {
    // ...
    pod.ownerEmissionsClaimed = true; // @audit single bool for both tokens

    if (reppoEmissionsPerEpoch > 0) {
        // ...
        if (podOwnerEmissionsShare > reppoSeedingBalance) {
            revert InsufficientReppoSeedings(); // @audit reverts entire tx
        }
        // ... transfer REPPO ...
    }

    if (primaryTokenEmissionsPerEpoch > 0) {
        // ...
        if (podOwnerEmissionsShare > primaryTokenSeedingBalance) {
            revert InsufficientPrimaryTokenSeedings(); // @audit reverts entire tx
        }
        // ... transfer primary token ...
    }
}
```

The same pattern exists in `claimVoterEmissions()`, which uses a single `voterHasClaimed` bool for both token types:

PodManagerV2.sol#L242-L356

```
function claimVoterEmissions(address voter, uint256 podId, uint256 epoch) external {
    // ...
    pod.voterHasClaimed[voter] = true; // @audit single bool for both tokens

    if (netPositive) {
        if (reppoEmissionsPerEpoch > 0) {
            // ...
            if (voterEmissionsShare > reppoSeedingBalance) {
                revert InsufficientReppoSeedings(); // @audit reverts entire tx
            }
            // ...
        }
        if (primaryTokenEmissionsPerEpoch > 0) {
            // ...
            if (voterEmissionsShare > primaryTokenSeedingBalance) {
                revert InsufficientPrimaryTokenSeedings(); // @audit reverts entire
                ↪  tx
            }
            // ...
        }
    }
    // ... same pattern for netNegative ...
}
```

The `EpochPod` struct uses a single boolean per claim type, offering no way to track REPPO and primary token claims independently:

PodManagerV2.sol#L45-L52

```
struct EpochPod {
    bool ownerEmissionsClaimed;
    uint256 totalUpVotes;
    uint256 totalDownVotes;
    mapping(address voter => bool) voterHasClaimed;
    mapping(address voter => uint256 votes) upVotesByVoter;
    mapping(address voter => uint256 votes) downVotesByVoter;
}
```

Consider a subnet where both `reppoEmissionsPerEpoch > 0` and `primaryTokenEmissionsPerEpoch > 0`. If the REPPO seed is well-funded but the primary token seed is depleted, every call to `claimPodOwnerEmissions()` or `claimVoterEmissions()` will revert on the `InsufficientPrimaryTokenSeedings` check. The claimant cannot receive their REPPO emissions even though sufficient REPPO seed exists. This persists until someone deposits enough primary token seed, which may never happen.

Impact

Earned emissions for one token type become indefinitely unclaimable when the other token type's seed is insufficient. Pod owners and voters who have legitimately earned rewards are blocked from receiving any of them, with no workaround available. The issue is exacerbated because there is no guarantee or obligation for anyone to re-seed either token, meaning rewards could be frozen permanently.

Recommendation

Split the claim tracking booleans in `EpochPod` so that REPPO and primary token emissions can be claimed independently:

```
struct EpochPod {
    bool ownerReppoEmissionsClaimed;
    bool ownerPrimaryTokenEmissionsClaimed;
    uint256 totalUpVotes;
    uint256 totalDownVotes;
    mapping(address voter => bool) voterReppoHasClaimed;
    mapping(address voter => bool) voterPrimaryTokenHasClaimed;
    mapping(address voter => uint256 votes) upVotesByVoter;
    mapping(address voter => uint256 votes) downVotesByVoter;
}
```

Then, instead of reverting when a seed is insufficient, skip that branch and allow the other token to be claimed. The skipped token can be claimed later once its seed is replenished.

Discussion

cartmanBruh96

This decision was based on giving a better UX for the end users (them being able to claim with one tx, and not have to do 2 txs) Also it is the responsibility of the subnet owner to honour the agreement (if the subnet owner has mentioned that his subnet has emissions in USDC, its his responsibility to ensure the sufficient balance exists) Failure to do so will be noted by community, and such bad actors's subnets will not have continuous interest from the users.

Issue M-8: Emission and Earnings Claims Use Current Parameters Instead of Epoch-Snapshotted Values [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/15

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`claimPodOwnerEmissions()`, `claimVoterEmissions()`, `claimReppoEarnings()`, and `claimPrimaryTokenEarnings()` all read mutable configuration parameters at claim time rather than values snapshotted when the epoch was active. If any of these parameters are updated between when an epoch ends and when users claim, claimers receive incorrect amounts.

Vulnerability Detail

When a user claims emissions for a past epoch, `PodManagerV2` calls `SubnetManager` getters to determine how much to distribute. These getters return the current mutable value from `SubnetManager.meta[subnetId]`, not the value that was in effect during the epoch being claimed.

In `claimPodOwnerEmissions()`:

PodManagerV2.sol#L176-L177

```
uint256 reppoEmissionsPerEpoch =
↪  $.subnetManager.getReppoEmissionsPerEpoch(subnetId);
uint256 primaryTokenEmissionsPerEpoch =
↪  $.subnetManager.getPrimaryTokenEmissionsPerEpoch(subnetId);
```

The same pattern in `claimVoterEmissions()`:

PodManagerV2.sol#L258-L259

```
uint256 reppoEmissionsPerEpoch =
↪  $.subnetManager.getReppoEmissionsPerEpoch(subnetId);
uint256 primaryTokenEmissionsPerEpoch =
↪  $.subnetManager.getPrimaryTokenEmissionsPerEpoch(subnetId);
```

These values are sourced from a single mutable field in `SubnetManager` that the subnet owner can update at any time:

SubnetManager.sol#L240-L257

```
function setReppoEmissionsPerEpoch(
    uint256 subnetId,
```

```
    uint256 newEmissions
) public onlySubnetOwner(subnetId) {
    SubnetStorage storage $ = _getSubnetStorage();
    $.meta[subnetId].reppoEmissionsPerEpoch = newEmissions;
    emit ReppoEmissionsPerEpochUpdated(subnetId, newEmissions);
}

function setPrimaryTokenEmissionsPerEpoch(
    uint256 subnetId,
    uint256 newEmissions
) public onlySubnetOwner(subnetId) {
    SubnetStorage storage $ = _getSubnetStorage();
    $.meta[subnetId].primaryTokenEmissionsPerEpoch = newEmissions;
    emit PrimaryTokenEmissionsPerEpochUpdated(subnetId, newEmissions);
}
```

The `Epoch` struct in `PodManagerV2` only stores vote data (up/down votes per pod, per voter) – it never records the emissions rate that was active during that epoch. No snapshotting occurs at epoch boundaries.

Consider the following scenario:

1. Epoch 5 runs with `reppoEmissionsPerEpoch = 1000e18`. Pod owners and voters participate based on this expected reward.
2. Epoch 5 ends. Before users claim, the subnet owner calls `setReppoEmissionsPerEpoch(subnetId, 100e18)` – reducing emissions by 90%.
3. Users claiming for epoch 5 now receive emissions calculated with `100e18` instead of the `1000e18` that was in effect when they participated.

The reverse is also problematic – if emissions are increased, claimers for old epochs receive more than intended, potentially draining the seeding balance and causing later claimers to hit `InsufficientReppoSeedings` reverts.

The same issue affects all of the following parameters, none of which are snapshotted per epoch:

1. **`reppoEmissionsPerEpoch`** – read live in `claimPodOwnerEmissions()` (PodManagerV2.sol#L176) and `claimVoterEmissions()` (PodManagerV2.sol#L258). Updatable via `setReppoEmissionsPerEpoch()` (SubnetManager.sol#L240). Directly controls the total emission amount distributed per epoch.
2. **`primaryTokenEmissionsPerEpoch`** – read live in `claimPodOwnerEmissions()` (PodManagerV2.sol#L177) and `claimVoterEmissions()` (PodManagerV2.sol#L259). Updatable via `setPrimaryTokenEmissionsPerEpoch()` (SubnetManager.sol#L250). Same impact as above for the primary token.
3. **`podOwnerEmissionSharePercentage`** – read live in `claimPodOwnerEmissions()` (PodManagerV2.sol#L196) and `claimVoterEmissions()` (PodManagerV2.sol#L284). Updatable via `setPodOwnerEmissionSharePercentage()` (SubnetManager.sol#L260).

Controls the owner/voter emission split – changing it retroactively redistributes funds between pod owners and voters.

4. `taxRate` – read live in `claimPodOwnerEmissions()` (PodManagerV2.sol#L200, PodManagerV2.sol#L220) and at multiple points in `claimVoterEmissions()` (PodManagerV2.sol#L299, L314, L333, L348). This is a global rate (not per-subnet) updatable by governance. Changing it retroactively alters the governance tax deducted from all unclaimed emission payouts.
5. `seederFeeShare` – read live in `claimReppoEarnings()` (SubnetManager.sol#L394) and `claimPrimaryTokenEarnings()` (SubnetManager.sol#L449). Updatable via `setSeederFeeShare()` (SubnetManager.sol#L273). Controls the owner/seeder split on subnet access fee earnings – changing it retroactively redistributes earnings between the subnet owner and external seeders for all unclaimed epochs.

Impact

Loss of funds for pod owners, voters, and external seeders. Any of the five parameters above can be altered after an epoch concludes, causing all claimers for that epoch to receive incorrect amounts. Increasing emissions retroactively can also drain the seeding pool, bricking claims for later epochs via `InsufficientReppoSeedings` reverts.

Recommendation

Snapshot all five claim-relevant parameters at the start or end of each epoch Claim functions should read from the snapshotted values for the relevant epoch rather than the current live state.

Discussion

cartmanBruh96

Same comment as for Issue 5. If subnet owner acts maliciously, they will be noted by the community, and the users will prevent participating in the subnet in the future. Given epochs are short (2 days), these kind of bad actors can be quickly detected, and the community will decide on the future participating based on these criteria

Issue L-1: `SubnetManager.increaseReppoEarnings()` and `SubnetManager.increasePrimaryTokenEarnings()` functions are never called [RESOLVED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/8

Vulnerability Detail

`SubnetManager.increaseReppoEarnings()` and `SubnetManager.increasePrimaryTokenEarnings()` functions have `onlyPodManagerV2` modifier but `PodManagerV2` contract never calls these functions:

```
modifier onlyPodManagerV2() {
    SubnetStorage storage $ = _getSubnetStorage();
    if (address($.podManager) != msg.sender) {
        revert CallerNotPodManagerV2();
    }
    _;
}
/// @inheritdoc ISubnetManager
function increaseReppoEarnings(
    uint256 subnetId,
    uint256 epoch,
    uint256 amount
) public onlyPodManagerV2 {
    SubnetStorage storage $ = _getSubnetStorage();
    $.earnings[subnetId].epochEarnings[epoch].reppoEarnings += amount;
}

/// @inheritdoc ISubnetManager
function increasePrimaryTokenEarnings(
    uint256 subnetId,
    uint256 epoch,
    uint256 amount
) public onlyPodManagerV2 {
    SubnetStorage storage $ = _getSubnetStorage();
    $.earnings[subnetId].epochEarnings[epoch].primaryTokenEarnings += amount;
}
```

Code Snippet

https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/SubnetManager.sol#L357-L374

Tool Used

Manual Review

Recommendation

Implement the logic to invoke both functions in `PodManagerV2` contract.

Issue L-2: Incorrect `baseURISubnetManager` points to veReppo metadata instead of subnet NFT metadata [RESOLVED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/9

Vulnerability Detail

The `baseURISubnetManager` points to metadata describing `veReppo` tokens, but this URI is used for `SubnetManager` NFTs which represent subnet ownership, causing incorrect metadata display in wallets and marketplaces.

```
string memory baseURISubnetManager =
    "https://ivory-fantastic-orangutan-61.mypinata.cloud/ipfs/bafkreifb4gc3vlumqxf4
    ↪  jtjzxyrixbd4brcomtwyhezvx2zven34rzvuka";
string memory baseURIPodManager =
    "https://ivory-fantastic-orangutan-61.mypinata.cloud/ipfs/bafkreibz6zh7g42aabll
    ↪  ajkp2greb26pi4k52uid3yor5zb4y2bre6rvu4";
```

Response from [baseURISubnetManager](https://ivory-fantastic-orangutan-61.mypinata.cloud/ipfs/bafkreifb4gc3vlumqxf4jtjzxyrixbd4brcomtwyhezvx2zven34rzvuka):

```
{
  "name": "veReppo",
  "external_url": "https://www.reppo.ai",
  "description": "A veReppo NFT represents your locked participation in the Reppo
  ↪  Network, granting you emissions power, governance influence, and a permanent
  ↪  mark of contribution to decentralized AI training. Each veReppo aligns with
  ↪  your voting epoch and determines your share of rewards across the Reppo
  ↪  ecosystem.",
  "image": "https://ivory-fantastic-orangutan-61.mypinata.cloud/ipfs/bafybeicmlmebf
  ↪  zo7b5kjfrvood3fiqqwtjgb2tr7kcl4bkzfg7ouul3exm"
}
```

Response from [`baseURIPodManager`](https://ivory-fantastic-orangutan-61.mypinata.cloud/ipfs/bafkreifb4gc3vlumqxf4jtjzxyrixbd4brcomtwyhezvx2zven34rzvuka](https://ivory-fantastic-orangutan-61.mypinata.cloud/ipfs/bafkreifb4gc3vlumqxf4jtjzxyrixbd4brcomtwyhezvx2zven34rzvuka%5D(https://ivory-fantastic-orangutan-61.mypinata.cloud/ipfs/bafkreibz6zh7g42aabllajkp2greb26pi4k52uid3yor5zb4y2bre6rvu4)):

```
{
  "name": "Reppo Pod",
  "external_url": "https://www.reppo.ai",
```

```
  "description": "A Reppo Pod NFT represents a human-AI collaboration published on
  ↪  the Reppo Network. Each Pod is a record of creative output, where voters
  ↪  provide preference feedback and generate alignment data that powers future AI
  ↪  training.  Pods can earn emissions, build reputation, and evolve across
  ↪  subnets.",
  "image": "https://ivory-fantastic-orangutan-61.mypinata.cloud/ipfs/bafybeihv5am6v
  ↪  wwiweoj7n4b6qwafafzi3rmhwav6rbai4shst44ft6rk4"
}
```

Code Snippet

[https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/src/SubnetManager.sol#L357-L374](https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/blob/main/economic-contracts/script/DeployV2.s.sol#L34-L37)

Tool Used

Manual Review

Recommendation

Updating the URL to point to correct `SubnetManager` and `PodManagerV2` metadata.

Issue L-3: hasSubnetAccess() Returns Stale true After Subnet Deletion [RESOLVED]

Source: https://github.com/sherlock-audit/2026-03-reppo-mar-13th-2026/issues/10

Summary

When a subnet is deleted via `deleteSubnet()`, the nested `mapping(address => bool) access` inside `MetaData` is not cleared due to a Solidity limitation. As a result, `hasSubnetAccess()` continues to return `true` for users who previously had access to the now-deleted subnet.

Vulnerability Detail

`deleteSubnet()` uses `delete $.meta[subnetId]` to remove the subnet metadata:

SubnetManager.sol#L286-L297

```
function deleteSubnet(uint256 subnetId) public onlySubnetOwner(subnetId) {
    SubnetStorage storage $ = _getSubnetStorage();
    uint256 lockedFee = $.lockedFees[subnetId];
    if (lockedFee > 0) {
        $.reppo.safeTransfer(msg.sender, lockedFee);
    }
    delete $.meta[subnetId]; // @audit does not clear nested mapping
    delete $.earnings[subnetId];
    delete $.lockedFees[subnetId];
    _burn(subnetId);
    emit SubnetDeleted(subnetId, lockedFee);
}
```

In Solidity, `delete` on a struct zeroes all value-type fields but **cannot** reset contained mappings because their keys are unknown at runtime. The `MetaData` struct contains a nested mapping:

ISubnetManager.sol#L8-L12

```
struct MetaData {
    address primaryToken;
    mapping(address => bool) access; // not cleared by delete
    uint256 accessFeeREPPO;
    // ...
}
```

After deletion, `hasSubnetAccess()` still reads the stale mapping entry and returns `true`:

SubnetManager.sol#L487-L490

```
function hasSubnetAccess(uint256 subnetId, address address_) public view returns
↪  (bool) {
    SubnetStorage storage $ = _getSubnetStorage();
    return $.meta[subnetId].access[address_];
}
```

Note that state-changing functions like `accessSubnetWithREPPOFee()` and `accessSubnetWithPrimaryTokenFee()` check `validSubnet()` before proceeding, and subnet IDs are monotonically incremented so they are not reused. This limits the practical impact to stale view-level data that could mislead off-chain integrations or frontends relying solely on `hasSubnetAccess()`.

Recommendation

Add a `validSubnet()` check to `hasSubnetAccess()` so it returns `false` for deleted subnets:

```
function hasSubnetAccess(uint256 subnetId, address address_) public view returns
↪  (bool) {
    if (!validSubnet(subnetId)) {
        return false;
    }
    SubnetStorage storage $ = _getSubnetStorage();
    return $.meta[subnetId].access[address_];
}
```

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.