yAudit

June 2026

Prepared for
Convex

Audited by
Ret2basic
HHK

Convex - Onchain Voting

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Convex Onchain Voting provides multiple smart contracts to execute DAO and Gauge voting onchain for Convex integration such as Curve, Fx and Resupply.

1.2 Audit Scope

This audit covers 17 smart contracts totaling approximately 1800 lines of code across 9 days of review.

```
src
├── ConvexCore.sol
├── CurveDaoProposer.sol
├── CurveGaugeExecutor.sol
├── CurveGaugeRegistry.sol
├── CurveVoteExecutor.sol
├── DaoVotePlatform.sol
├── Delegation.sol
├── FxGaugeExecutor.sol
├── FxGaugeRegistry.sol
├── GaugeList.sol
├── GaugeProposer.sol
├── GaugeVoteHelper.sol
├── GaugeVotePlatform.sol
├── GenericDaoProposer.sol
├── ResupplyDaoProposer.sol
├── ResupplyVoteExecutor.sol
├── SurrogateRegistry.sol
└── VotingRegistry.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	5
Low	1
Informational	12

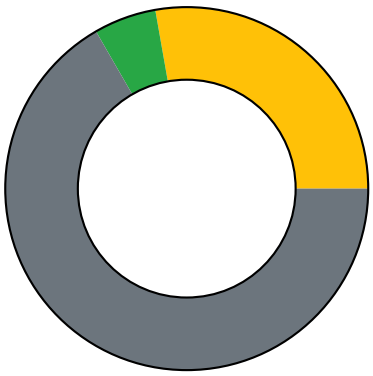

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The Convex Voting Platform aggregates vlCVX weight, applies lazy delegation, and forwards consolidated votes to upstream systems (Curve, F(x), Frax, Resupply, Convex). The risk concentrated in the custom lazy-delegation accounting (per-epoch packed weights in assembly, deferred snapshots, and the ordering of syncs against votes), where nearly all major findings originated. The team was highly responsive, fixing the significant issues promptly and acknowledging the rest with sound rationale.

2 Audit Overview

2.1 Project Information

Protocol Name: Convex
Repository:
https://github.com/convex-eth/voting/tree/8a6d1d243feaa0d786263f503cfce96459423aaf
Commit Hash: 8a6d1d243feaa0d786263f503cfce96459423aaf
Commit URL:
https://github.com/convex-eth/voting/tree/8a6d1d243feaa0d786263f503cfce96459423aaf

2.2 Audit Timeline

The audit was conducted from June 1 to 11, 2026.

2.3 Audit Resources

- Code repositories and documentation
- Audit document

Category	Mark	Description
Access Control	Good	Consistently structured with Ownable2Step, operator-gated entrypoints, and a surrogate/accepted-signer model.
Mathematics	Average	Clean and simple mathematics throughout the codebase.
Complexity	Average	Some complexity around the lazy delegation layer, snapshots, and per-epoch weights, which resulted in multiple findings.
Libraries	Good	Use of appropriate and battle-tested libraries such as OpenZeppelin.
Decentralization	Average	Some centralization: operators get arbitrary execute, and the owner can seed delegations, force-remove gauges, and set operators.
Documentation	Average	Helpful documentation and natspec for most of the codebase.
Testing and verification	Average	Broad unit, fuzz, invariant, and fork tests, though some findings could have been caught by a more complete test suite.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

2.6.1 Delegate can reclaim a direct voter's relocked weight before the voter re-votes

When a delegator votes directly before their delegate votes, `GaugeVotePlatform._initBaseInfo` removes the delegator's then-current delegated weight from the delegate's `adjustedWeight`. A later permissionless `Delegation.syncAtEpoch` can raise that delegator's proposal-epoch weight (via a relock into the epoch), and when the delegate then votes for the first time, `_initBaseInfo` adds the current `balanceAtEpochOf(epoch, delegate)`, which now includes the increase. The delegate keeps the delta even though the delegator already voted directly.

Technical Details

The removal at the direct vote uses the `del.voteStatus == 0` branch (`-= userWeightAtEpochOf(epoch, user)`): a fixed pre-charge of the pre-relock amount. It is not nonce-aware, and the delegate's own init later adds the post-relock balance with no offsetting correction, so the delegate nets the relock delta.
Conditions: the user delegated to the delegate, voted directly before the delegate voted, then relocked expired vlCVX into the proposal epoch, with no prior sync snapshot for `(user, epoch)`. Since `syncAtEpoch` is permissionless, the delegate needs no cooperation from the user. This affects both `DaoVotePlatform._initBaseInfo` and `GaugeVotePlatform._initBaseInfo`.
This is distinct from #2: there the delegate votes first and the nonce branch governs removal; here the delegator votes first (hitting the nonce-free pre-charge) and the delegate self-init adds the grown balance, a path the nonce comparison never reaches.
`GaugeVoteHelper.getContributingWeights` returns 0 for the directly-voted user, so the on-chain skew is not visible through that view either (related to #14).

Impact

Medium. The delegate reclaims the delegator's post-vote relock delta and can redirect it to other gauges or flip a DAO outcome, against the direct voter's expressed preference. PoC `test/RelockDeltaDelegatePoC.t.sol`: after Alice votes directly with 50 and relocks to 150, an external `syncAtEpoch` lets Bob vote with 150 effective weight while Alice's 50 still counts (gauge: 50 on Alice's gauge vs 150 on Bob's; DAO: flips from no=50 to yes=150, no=50). Bounded to the delegator's own post-vote relock increase, no fund loss or inflation.

Recommendation

Once a user has voted directly on a proposal at epoch E, stop crediting later delegated growth at E to the delegate. Either ignore `syncAtEpoch(user, E)` increases in `_initBaseInfo` / `_vote`, or record an equal-and-opposite `pendingWeightAdjustment` for the delegate when a sync grows a directly-voted delegator. Apply to both platforms.

Developer Response

Acknowledged. I don't think you could even effectively guard against this since the delegate doesn't know who or how many people influenced the growth in delegation power.

2.6.2 Late permissionless gauge proposal creation lets a small voter define the full gauge allocation round

`GaugeProposer.proposeVote()` is permissionless and creates each biweekly gauge proposal using an epoch-derived `startTime` and `endTime`, not the call time. If no honest caller opens the round near the epoch start, an attacker can wait until the final seconds before `epochStart + proposalLength`, open the only proposer-backed gauge round for that epoch, and immediately vote with a small amount of vlCVX. Because gauge voting has no quorum, that small late vote can become 100% of the finalized local gauge result and can be relayed upstream as the full Convex gauge allocation for the round.

Technical Details

The gauge proposer is intentionally public:

```
// src/GaugeProposer.sol
function proposeVote() external {
    vlCVX.checkpointEpoch();

    uint256 ec = vlCVX.epochCount();
    require(ec % 2 == 0, "Must be even epoch (bi-weekly)");
    require(ec > lastEpochUsed, "Epoch already used");

    lastEpochUsed = ec;

    uint256 currentEpoch = ec - 2;
    (, uint32 epochStart) = vlCVX.epochs(currentEpoch);

    uint256 startTime = uint256(epochStart);
    uint256 endTime = startTime + proposalLength;

    uint256 proposalId = gaugeVotePlatform.proposalCount();

    gaugeVotePlatform.createProposal(startTime, endTime);

    emit GaugeVoteProposed(proposalId);
}
```

`GaugeVotePlatform.createProposal()` checks that the stored proposal duration is within bounds and that `endTime > block.timestamp`, but it does not enforce a minimum remaining live voting window:

```
// src/GaugeVotePlatform.sol
function createProposal(uint256 _startTime, uint256 _endTime) public onlyOperator {
    uint256 pCnt = proposals.length;
    if (pCnt > 0) {
        if (block.timestamp <= proposals[pCnt - 1].endTime + overtime) revert
PrevNotEnded();
    }

    if (_endTime <= _startTime) revert BadTime();
    if (_endTime <= block.timestamp) revert BadTime();
    if (_endTime - _startTime < MIN_PROPOSAL_DURATION) revert BadTime();
    if (_endTime - _startTime > MAX_PROPOSAL_DURATION) revert BadTime();

    vlCVX.checkpointEpoch();
    uint256 epoch = vlCVX.epochCount() - 2;

    proposals.push(Proposal({
        startTime: uint48(_startTime),
        endTime: uint48(_endTime),
        epoch: uint48(epoch)
    }));
    emit NewProposal(proposals.length - 1, _startTime, _endTime);
}
```

Gauge voting itself has no quorum. The executor later normalizes whatever finalized local gauge totals exist to 10,000 BPS:

```
// src/CurveGaugeExecutor.sol
uint256 totalVotes = votePlatform.voteTotals(proposalId);
...
uint256 gt = votePlatform.gaugeTotal(proposalId, gauges[i]);
weights[i] = gt * WEIGHT_BPS / totalVotes;
```

Therefore, if the attacker is the only participant during the compressed tail window, their gauge receives the full external allocation:

```
attacker voteTotals = 1 * WEIGHT_DIVISOR
attacker gaugeTotal = 1 * WEIGHT_DIVISOR
submitted weight = gaugeTotal * 10000 / voteTotals = 10000
```

The deployment wires the permissionless proposer as an operator on both Curve and F(x) gauge voting platforms:

```
// script/Deploy.s.sol
core.execute(address(curveGaugeVoting), abi.encodeWithSignature(
    "setOperator(address,bool)", address(curveGaugeProposer), true
));

core.execute(address(fxGaugeVoting), abi.encodeWithSignature(
    "setOperator(address,bool)", address(fxGaugeProposer), true
));
```

Impact

An untrusted caller can deliberately wait until the proposer-backed gauge round has almost no voting time left, open that round, and use a small amount of vlCVX to determine the entire submitted external gauge allocation if other voters do not react inside the compressed window. This can redirect Curve or F(x) gauge weights for the affected round and therefore affect downstream gauge emissions and governance-controlled allocation outcomes.

The issue is bounded to one gauge round and requires the round not to have been opened earlier, so it does not justify High severity. However, the attacker-controlled timing, permissionless trigger, lack of quorum, and full external allocation effect support Medium severity rather than Low.

Recommendation

Require `GaugeProposer.proposeVote()` to be called within a bounded interval after the target epoch starts. If that interval is missed, the proposer-backed round should be skipped or require an operator/guardian recovery path with an explicit minimum remaining voting window. Do not allow a public caller to create the only scheduled gauge round for an epoch when less than the configured minimum live voting time remains.

Developer Response

Acknowledged. Because there is no quorum and not a governance vote, this is not an issue for me. If no one shows up to vote then the attacker is the only voter and gets all the weight.

2.6.3 Delegation sync skips the 17th live vlCVX lock record

`Delegation._syncUser` reconstructs delegated weights over a 16-epoch fill window but scans only 16 `userLocks` records starting at `nextUnlockIndex`. The upstream vlCVX locker can contain 16 active weekly locks plus a fresh pending next-epoch lock. The upstream `balanceAtEpochOf` iterates the full lock array, so Delegation can omit the pending 17th record from future delegated weights.

Technical Details

Delegation allocates a 17-slot buffer but scans only 16 lock records:

```
// src/Delegation.sol
uint256[17] memory epochWeights;

if (_locked > 0) {
    for (uint256 i = nextUnlockIndex; i < nextUnlockIndex + 16;) {
        ...
        epochWeights[e - _startEpoch] += _lockBoosted;
        unchecked { ++i; }
    }
}
```

The downloaded vlCVX locker's `balanceAtEpochOf` traverses the user's lock array and includes all locks whose lock epoch contributes to the queried epoch:

```
// integrated_code/convex-platform/contracts/contracts/CvxLockerV2.sol
for (uint i = locks.length - 1; i + 1 != 0; i--) {
    uint256 lockEpoch = uint256(locks[i].unlockTime).sub(lockDuration);
    if (lockEpoch <= epochTime) {
        if (lockEpoch > cutoffEpoch) {
            amount = amount.add(locks[i].boosted);
        } else {
            break;
        }
    }
```

```
11 }
```

Fresh non-relock locks are assigned to the next weekly epoch, creating a pending next-epoch record in addition to the current 16 active records.

Impact

A maximally laddered locker that delegates can have its newest pending weekly lock omitted from delegated future-epoch accounting. The undercount persists until a later sync path corrects the affected epoch or the delegator participates directly. If multiple such delegators point at the same delegate, their suppressed pending-lock weight aggregates on that single delegate and can materially reduce the delegate's effective proposal-epoch voting power. In close DAO or gauge votes, that accumulated undercount can distort or even flip the resulting tally. This is still an undercount rather than direct asset loss, but it is a real governance-integrity issue rather than a trivial edge-case rounding loss.

Recommendation

Scan enough lock records to cover the 16 active records plus a pending next-epoch record, or derive delegated weights directly from `vlCVX.balanceAtEpochOf` for each filled epoch.

Developer Response

Fixed in commit ca1ddb2a4ba60f2c77848f865123b06dab7eff2e.

2.6.4 Same-block delegation sync ordering can double-count relocked vlCVX voting weight

Delegation sync snapshots record only `block.timestamp`. The voting platforms decide whether a delegate voted before or after a sync by checking whether `lastVoteTime > snapshot.timestamp`. If sync and delegate vote happen in the same block timestamp, equality is treated as the before-sync branch. A delegate can keep newly synced delegated weight in their vote, while the delegatee also votes directly with that same relocked weight, inflating local DAO and gauge vote totals above real proposal-epoch vlCVX backing.

Technical Details

The root cause is not that `Delegation` fails to record enough information about weight changes, nor that the voting platforms forgot to implement delegated-weight removal. Both DAO and Gauge voting do attempt to remove a delegatee's weight from an already-cast delegate vote. The bug is narrower and more dangerous: the removal logic uses a strict `>` comparison on timestamps, so the `lastVoteTime == snapTs` case is silently treated as if the delegate had voted before the sync.
The overall state machine is supposed to work like this:

1. A proposal is created and permanently bound to a fixed `proposalEpoch`.
2. Alice has delegated to Bob, so Bob's live vote may include Alice's delegated weight for that epoch.

3. Alice later causes her weight in that same `proposalEpoch` to increase by relocking expired vlCVX and syncing the delegation book.
4. If Bob had already voted with the post-sync weight included, Alice's subsequent direct vote must first remove that same delegated weight from Bob's already-cast vote.

That last removal step is where the bug lives.
First, note that proposal weight is always read from a fixed epoch chosen when the proposal is created:

```
// src/DaoVotePlatform.sol
function createProposal(uint256 _startTime, uint256 _endTime, VoteType _voteType,
uint256 _proposalId) public onlyOperator {
    ...
    vlCVX.checkpointEpoch();
    uint256 epoch = vlCVX.epochCount() - 2;

    proposals.push(Proposal({
        startTime: uint48(_startTime),
        endTime: uint48(_endTime),
        epoch: uint48(epoch),
        voteType: uint8(_voteType),
        proposalId: uint104(_proposalId)
    }));
}
```

This means the exploit does not require the attacker to create a proposal after expiry; the real requirement is that there exists a target proposal whose fixed `proposalEpoch` is exactly the epoch in which Alice's relock causes a `0 -> positive` weight change.
Second, the upstream vlCVX implementation makes relock count toward the current epoch rather than the next one. That is what creates the exploitable `0 -> positive` jump in proposal-epoch weight:

```
// integrated_code/convex-platform/contracts/contracts/CvxLockerV2.sol
uint256 lockEpoch = block.timestamp.div(rewardsDuration).mul(rewardsDuration);
if (!_isRelock) {
    lockEpoch = lockEpoch.add(rewardsDuration);
}

uint256 eIndex = epochs.length - 1;
if (_isRelock) {
    eIndex--;
}
Epoch storage e = epochs[eIndex];
e.supply = e.supply.add(uint224(boostedAmount));
```

So, if Alice's old lock has expired, a fresh lock would only affect the next epoch, but a relock updates the current epoch. For a proposal already bound to that epoch, Alice's real proposal-epoch backing can jump from `0` to `1000 * 1e17`.
Third, `Delegation.sync()` stores only two pieces of ordering information: the old packed weight and the current timestamp.

```
// src/Delegation.sol
struct SyncSnapshot {
    uint96 timestamp;
    uint32 preSyncWeight;
}

syncSnapshots[_user][_epoch] = SyncSnapshot({
    preSyncWeight: uint32(prePacked),
    timestamp: uint96(block.timestamp)
});
```

For the exploit path, the important case is:

- `preSyncWeight = 0`
- `currentDelWeight = 1000 * 1e17`
- `snapTs = t`

Bob then votes in the same `block.timestamp` as that sync. His vote therefore includes Alice's freshly synced delegated weight, and his `lastVoteTime` is also recorded as `t`.
When Alice later votes, the platform tries to remove Alice's delegated weight from Bob's already-cast vote. Both DAO and Gauge implementations classify the ordering using this strict comparison:

```
// src/DaoVotePlatform.sol and src/GaugeVotePlatform.sol
if (snapTs > 0 && uint256(del.lastVoteTime) > snapTs) {
    weightToRemove = int256(currentDelWeight);
} else if (snapTs > 0) {
    weightToRemove = int256(snapWeight);
    int256 diff = int256(currentDelWeight) - int256(snapWeight);
    if (diff > 0) {
        pendingWeightAdjustment[_proposalId][delegate] -= int96(diff);
    }
}
```

This is not limited to one code path. The same timestamp-based classification is present in:

1. `DaoVotePlatform._initBaseInfo()` when removing a delegatee's weight from an already-cast delegate vote.
2. `DaoVotePlatform._vote()` when a later resync reveals additional delegated weight delta during a revote.
3. `GaugeVotePlatform._initBaseInfo()` for the same delegate-removal path.
4. `GaugeVotePlatform._vote()` for the same resync/revote path.

So the vulnerability simultaneously affects both governance lines that consume `Delegation` state. Fixing only `DaoVotePlatform` or only `GaugeVotePlatform` would leave the other path exploitable.
The intended meaning of the branches is:

- `del.lastVoteTime > snapTs`: Bob voted after the sync, so his live vote already contains the post-sync delegated weight. Remove the full `currentDelWeight` immediately.
- `else if (snapTs > 0)`: Bob voted before the sync, so his live vote should only contain the pre-sync amount. Remove only `snapWeight`, and defer only the newly added delta.

But the equality case is not modeled. If `del.lastVoteTime == snapTs`, the code falls through into the second branch and treats the vote as if it had happened before the sync.
That is exactly wrong for this path. In the exploit scenario:

- `snapWeight = 0`
- `currentDelWeight = 1000 * 1e17`
- `del.lastVoteTime = snapTs = t`

So the code removes only `snapWeight`, which is zero, and pushes the whole `1000 * 1e17` delta into `pendingWeightAdjustment` instead of deducting it from Bob's current live vote.
On DAO voting, this means Bob's yes/no totals remain overstated immediately after Alice votes.

```
// src/DaoVotePlatform.sol
weightToRemove = int256(snapWeight);
int256 diff = int256(currentDelWeight) - int256(snapWeight);
if (diff > 0) {
    pendingWeightAdjustment[_proposalId][delegate] -= int96(diff);
}

_changeVoteTotals(_proposalId, -weightToRemove, del.yesWeight, del.noWeight);
```

On Gauge voting, the same misclassification leaves both `voteTotals` and the per-gauge totals inflated, because the immediate per-gauge removal loop also runs with the wrong `weightToRemove`.

```
// src/GaugeVotePlatform.sol
for (uint256 i = 0; i < len;) {
    _changeGaugeTotal(
        _proposalId,
        delegateVotes[i].gauge,
        -(int256(uint256(delegateVotes[i].weight)) * weightToRemove / int256(max_weight)
)
    );
    unchecked { ++i; }
}

voteTotals[_proposalId] -= uint256(weightToRemove);
```

The dangerous part is that this is not a harmless temporary mismatch. The removal is deferred, not automatically settled. If Bob does not cast another vote on that proposal, the negative `pendingWeightAdjustment` is never consumed, so Bob's already-recorded vote keeps Alice's delegated weight while Alice's own vote adds the same weight again.
That is why the same underlying proposal-epoch backing can end up counted twice:

1. once inside Bob's already-cast delegate vote,
2. once again inside Alice's direct vote.

The result is a local tally that can exceed real proposal-epoch vlCVX backing. On DAO proposals this distorts yes/no totals and quorum calculations. On gauge proposals it distorts both the aggregate total and the per-gauge allocations.

Impact

An account controlling both a delegatee and its delegate can count relocked proposal-epoch weight twice in the same block. The exploit conditions are narrow, but once hit, the bug corrupts the proposal's local accounting rather than causing a harmless transient mismatch.
DAO impact:

1. `DaoVotePlatform` can overstate yes/no totals for the affected proposal.
2. That can make quorum appear satisfied with less real vlCVX participation than intended.
3. It can also skew the final yes/no ratio used to judge the local governance result.

Gauge impact:

1. `GaugeVotePlatform` can overstate the aggregate `voteTotals` for the proposal.
2. It can also overstate the totals of the specific gauges that were already present in the delegate's recorded vote.

3. That means the bug is not just a global accounting issue; it can distort the relative distribution across gauges and therefore the outbound allocation signal.

The inflation is bounded by the synced delegated-weight delta and still requires the same-timestamp sync/vote path, but it directly breaks voting-accounting integrity in whichever platform remains unpatched.

Recommendation

Do not infer sync-versus-vote ordering from `block.timestamp`. Instead, add proposal-local delegated-weight accounting in both `DaoVotePlatform` and `GaugeVotePlatform`, keyed by `(proposalId, delegator)`, to record how much of that delegator's weight has already been incorporated into the proposal's live vote state.
Whenever either the delegate or the delegator interacts with a proposal, read `delegation.userWeightAtEpochOf(proposalEpoch, delegator)`, compare it against the proposal-local accounted amount, compute the delta, apply that delta once to the delegate-side live accounting, and then update the recorded amount. In DAO this must update the delegate's yes/no totals and `adjustedWeight`; in Gauge it must update the per-gauge totals, aggregate `voteTotals`, and `adjustedWeight`.
Apply the same settlement logic in both `_initBaseInfo()` and `_vote()`, and in both platforms. Simply changing `>` to `>=` is not sufficient, because it only flips the ambiguous same-timestamp case. As a short-term mitigation, the protocol could revert when `lastVoteTime == snapTs`, but the proper fix is to replace timestamp inference with explicit proposal-local settlement.

Appendix: PoC

Save the following code as `test/poc/SameBlockDelegationSyncDoubleCountsRelockedWeightPoC.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../../src/DaoVotePlatform.sol";
import "../../src/SurrogateRegistry.sol";
import "../../src/Delegation.sol";
import "../mocks/simpleVlCvx.sol";

contract SameBlockDelegationSyncDoubleCountsRelockedWeightPoC is Test {
    simpleVlCvx internal vlcvx;
    Delegation internal delegation;
    SurrogateRegistry internal surrogateRegistry;
    DaoVotePlatform internal dao;

    address internal alice = makeAddr("alice");
    address internal bob = makeAddr("bob");
    address internal operator = makeAddr("operator");

    uint256 internal constant WEEK = 7 days;
    uint256 internal constant WD = 1e17;

    function setUp() public {
        vm.warp(1700000000);

        vlcvx = new simpleVlCvx();
        delegation = new Delegation("Convex Delegation", address(this), address(vlcvx));
```

```
28          surrogateRegistry = new SurrogateRegistry("Convex Surrogate Registry");
29
30          dao = new DaoVotePlatform(
31              "Convex Dao Voting",
32              address(this),
33              address(vlcvx),
34              address(surrogateRegistry),
35              address(delegation)
36          );
37
38          dao.setOperator(operator, true);
39      }
40
41      function testPoC_SameBlockDelegationSyncDoubleCountsRelockedWeight() public {
42          _lockAndDelegate(alice, 1000, bob);
43
44          uint256 unlockTime = ((vm.getBlockTimestamp() / WEEK) * WEEK) + WEEK + 16 * WEEK;
45          vm.warp(unlockTime + 1);
46
47          uint256 pid = _createProposalForCurrentEpoch();
48          (,, uint48 proposalEpoch,,) = dao.proposals(pid);
49
50          vm.prank(alice);
51          vlcvx.relock(alice, 1000 * WD, 0);
52          delegation.sync(alice);
53
54          (uint256 snapWeight, uint256 snapTs) = delegation.getSyncSnapshot(alice,
    proposalEpoch);
55          assertEq(snapWeight, 0);
56          assertEq(snapTs, vm.getBlockTimestamp());
57
58          _voteYes(bob);
59          _voteYes(alice);
60
61          uint256 realProposalWeight = vlcvx.balanceAtEpochOf(proposalEpoch, alice)
62              + vlcvx.balanceAtEpochOf(proposalEpoch, bob);
63          assertEq(realProposalWeight, 1000 * WD);
64          assertEq(dao.voteTotals(pid), 2000 * WD);
65          assertGt(dao.voteTotals(pid), realProposalWeight);
66      }
67
68      function _lockAndDelegate(address user, uint256 amount, address delegateAddr)
    internal {
69          vlcvx.lock(user, amount * WD, 0);
70          vm.prank(user);
71          delegation.setDelegate(delegateAddr);
72      }
73
74      function _createProposalForCurrentEpoch() internal returns (uint256 pid) {
75          uint256 startTime = vm.getBlockTimestamp();
76          uint256 endTime = startTime + 1 days;
77          vm.prank(operator);
78          dao.createProposal(startTime, endTime, DaoVotePlatform.VoteType.Parameter, 1);
79          pid = dao.proposalCount() - 1;
80      }
81
82      function _voteYes(address user) internal {
83          vm.prank(user);
84          dao.vote(user, 10000, 0);
85      }
86  }
```

The test uses a deterministic local setup to force the vulnerable path:

1. Alice delegates to Bob.
2. Alice's old lock expires.
3. A DAO proposal is created for the current epoch. In the PoC this proposal is created after expiry to pin `proposalEpoch` to the exact epoch where relock will restore weight; that is a harness convenience, not a requirement that the attacker must create proposals in production.
4. Alice relocks and calls `delegation.sync(alice)`, producing a snapshot with `preSyncWeight = 0`.
5. Bob votes in the same `block.timestamp` as the sync, so his vote includes Alice's newly synced delegated weight.
6. Alice votes before Bob revotes and consumes the pending adjustment.

The final assertion shows that the real proposal-epoch backing is only `1000 * 1e17`, but the local tally has become `2000 * 1e17` because Bob's vote kept Alice's delegated weight and Alice's direct vote added it again.
Run it with:

```
forge test --match-path test/poc/SameBlockDelegationSyncDoubleCountsRelockedWeightPoC.t.sol --match-test testPoC_SameBlockDelegationSyncDoubleCountsRelockedWeight -vvv
```

Developer Response

Fixed in commit 171c65cf798dbc65c402685eec563746d6be45ca and a299b7b2161a09822aef5dfb8d0cf253365ca4f2.

2.6.5 F(x) gauge executor calls an owner-only Booster entrypoint without an ownership handover path

`FxGaugeExecutor` treats the cvxFXN Booster as a generic gauge-voting endpoint and calls `voteGaugeWeight` directly after a local F(x) gauge round is finalized. In the upstream implementation, `voteGaugeWeight` is not a public relay hook; it is an `onlyOwner` management entrypoint on the Booster. Because `FxGaugeExecutor` is a standalone settlement contract, `msg.sender` at the Booster is always the executor itself, not the Booster owner, so finalized rounds revert with `!o_auth` unless the executor address has actually become the Booster owner. Although the Booster was deployed by Convex-controlled infrastructure and is therefore administratively fixable, the shipped executor has no code path to complete the Booster's two-step ownership handover. The default integration therefore cannot settle F(x) gauge votes without an additional adapter, helper, or redeployment flow.

Technical Details

The local voting layer can successfully finalize an F(x) gauge proposal and hand execution to `FxGaugeExecutor`. The executor checks the local proposal state, converts the finalized vote totals into basis-point weights, and then forwards those weights to a fixed external contract:

```
// src/FxGaugeExecutor.sol
address public constant gaugeController = address(0xe60eB8098B34eD775ac44B1ddE864e098C6d7f37);
```

```
address public constant gaugeVoter = address(0xAffe966B27ba3E4Ebb8A0eC124C7b7019CC762f8);


IFxGaugeVoter(gaugeVoter).voteGaugeWeight(gaugeController, gauges, weights);
```

At this point the external call is made by the executor contract itself. Even if an EOA or automation bot triggers `executeGaugeVote`, the upstream contract observes `msg.sender == address(FxGaugeExecutor)`.
The upstream F(x) Booster shows that the target function is owner-only:

```
// integrated_code/fx/contracts/contracts/Booster.sol
modifier onlyOwner() {
    require(owner == msg.sender, "!o_auth");
    _;
}

function voteGaugeWeight(
    address _controller,
    address[] calldata _gauge,
    uint256[] calldata _weight
) external onlyOwner {
    for (uint256 i = 0; i < _gauge.length;) {
        bytes memory data = abi.encodeWithSelector(
            bytes4(keccak256("vote_for_gauge_weights(address,uint256)")),
            _gauge[i],
            _weight[i]
        );
        _proxyCall(_controller, data);
        unchecked { ++i; }
    }
}
```

For `FxGaugeExecutor` to make this call successfully, the executor address itself must therefore become the Booster owner. The Booster code provides only two ways for an address to become `owner`.
First, the deployer becomes owner in the constructor:

```
// integrated_code/fx/contracts/contracts/Booster.sol
constructor(address _proxy, address _depositor, address _cvxfxn, address _poolReg,
address _feeReg) {
    proxy = _proxy;
    fxnDepositor = _depositor;
    cvxfxn = _cvxfxn;
    isShutdown = false;
    owner = msg.sender;
    rewardManager = msg.sender;
    poolManager = msg.sender;
    poolRegistry = _poolReg;
    feeRegistry = _feeReg;
}
```

Second, the current owner can nominate a pending owner, but the pending owner must then accept ownership from its own address:

```
// integrated_code/fx/contracts/contracts/Booster.sol
function setPendingOwner(address _po) external onlyOwner {
    pendingOwner = _po;
    emit SetPendingOwner(_po);
}

function acceptPendingOwner() external {
    require(pendingOwner != address(0) && msg.sender == pendingOwner, "!p_owner");
```

```
9       owner = pendingOwner;
10      pendingOwner = address(0);
11  }
```

No other code path in the Booster assigns `owner`. This is the central incompatibility: the already-deployed Booster's constructor path has already run, and the only remaining ownership path requires the pending owner to actively call `acceptPendingOwner()`.
The current `FxGaugeExecutor` cannot make that acceptance call. Its external surface is limited to gauge execution and read helpers; it does not expose an ownership-acceptance function, an admin-controlled arbitrary call function, or any other path that can call `Booster.acceptPendingOwner()` as `msg.sender == address(FxGaugeExecutor)`:

```
1  // src/FxGaugeExecutor.sol
2  function executeGaugeVote(uint256 proposalId, address[] calldata gauges) external {
3      ...

5      IFxGaugeVoter(gaugeVoter).voteGaugeWeight(gaugeController, gauges, weights);
6      emit GaugeVoteExecuted(proposalId, gauges, weights);
7  }
```

This creates a hard authorization mismatch in the current integration:

1. Local voting succeeds because `GaugeVotePlatform` only tracks local proposal finalization and per-gauge totals.
2. Settlement enters `FxGaugeExecutor.executeGaugeVote` and computes outbound weights.
3. The executor calls `Booster.voteGaugeWeight(...)`.
4. The Booster checks `owner == msg.sender`.
5. `msg.sender` is the executor contract, so the call reverts unless the executor itself is the Booster owner.

The fact that the Booster was deployed by a Convex-controlled deployer does not by itself make the current executor authorized. It means Convex likely controls the administrative side needed to remediate the integration, but the current execution path still needs the executor contract address to satisfy `owner == msg.sender` at the Booster.
The deployment wiring does not solve that mismatch. `script/Deploy.s.sol` deploys and registers `FxGaugeExecutor`, but it does not grant any upstream Booster authority:

```
1  // script/Deploy.s.sol
2  fxGaugeExecutor = new FxGaugeExecutor("FX Gauge Executor", address(fxGaugeVotePlatform));

3  votingRegistry.setVotingContract("FX", IVotingRegistry.Action.GAUGE_EXECUTOR, address(
   fxGaugeExecutor));
```

More importantly, the current executor cannot complete the ownership handover after deployment. The Booster owner can call `setPendingOwner(address(fxGaugeExecutor))`, but that only writes `pendingOwner`; it does not make the executor the owner. The second transaction must be `acceptPendingOwner()` with `msg.sender == address(fxGaugeExecutor)`. Since `FxGaugeExecutor` has no function that can perform that call, the current executor address cannot become owner through the existing code path. Remediation requires a changed executor, an adapter that can accept ownership and relay the vote, or another explicitly authorized upstream path.
The upstream source also warns that this Booster/operator address is replaceable and integrations should reference the current operator instead of hardcoding a specific Booster instance. That means the current design is brittle in two separate ways: it targets the wrong permission surface today, and it hardcodes an address the upstream system explicitly treats as replaceable.

Impact

Finalized F(x) gauge votes can pass local voting but fail at the only on-chain settlement step under the shipped wiring. Convex's aggregated F(x) gauge weights are therefore not submitted through this executor, leaving the F(x) relay path unavailable until Convex performs an additional ownership/adapter/redeployment fix. This is more severe than a missing one-step deployment setting because simply nominating the executor as `pendingOwner` is insufficient; the current executor code cannot complete the required acceptance transaction. This is a full platform-specific governance liveness failure in the default integration, not a direct custody loss.

Recommendation

Replace the direct hardcoded Booster call with a dedicated adapter that resolves the current authorized F(x) operator and submits gauge votes through an already valid upstream authority path. Only register `FxGaugeExecutor` once deployment checks confirm that the final caller seen by `voteGaugeWeight` is authorized in the live upstream configuration.

Developer Response

Fixed in commit cf664f59b5139abe0983349abbe37dc2da8a5da2.

2.7 Low Findings

2.7.1 First mirror caller can choose the vlCVX snapshot for permissionless DAO mirrors

`CurveDaoProposer.proposeVote` and `ResupplyDaoProposer.proposeVote` are permissionless and mirror an upstream DAO vote using `startTime = block.timestamp`, so `DaoVotePlatform.createProposal` snapshots `vlCVX.epochCount() - 2` at call time rather than at the upstream proposal's start. The first caller therefore chooses the snapshot epoch.

Technical Details

Both functions accept any caller within 3 days of the upstream open, then mark the vote id used (`ownershipProposalsUsed` / `parameterProposalsUsed` / `proposalsUsed`) so only the first mirror counts. If the upstream proposal opened within roughly 3 days of a weekly vlCVX epoch boundary, the first caller can wait until after the rollover and mirror in the next epoch, shifting the snapshot forward by one week. That changes the electorate (holders who locked, relocked, or let locks expire after the upstream open are included or excluded depending on when the mirror is created) and changes `vlCVX.totalSupplyAtEpoch(epoch)`, the quorum denominator.

Impact

Low. The chosen snapshot can change the local yes/no result and the quorum base, and because `CurveVoteExecutor` / `ResupplyVoteExecutor` relay the local result upstream, the distortion reaches the real governance action. It only applies to proposals opening near an epoch boundary, and because the entrypoint is permissionless, honest users or keepers are likely to mirror promptly, which leaves little practical room to delay. PoC

`test/CurveDaoProposerEpochShiftPoC.t.sol`: an immediate mirror excludes a late locker while a delayed mirror includes them and flips the outcome.

Recommendation

Derive the snapshot epoch and the local `startTime` / `endTime` from the upstream proposal's `startDate` / `createdAt` so a mirror always reflects the electorate at the upstream open, instead of from caller transaction time. If upstream timing metadata is insufficient to compute the epoch safely, gate mirroring to a trusted operator expected to mirror promptly.

Developer Response

Fixed in commit 312a15943817f1fadbd795ed1f8424def1d22133 & 01c879435e20ccac48eb9a36f33e9b17046b73d1.

2.8 Gas Savings Findings

None.

2.9 Informational Findings

2.9.1 Unused `_noWeight` parameter in `_changeVoteTotals`

Technical Details

`DaoVotePlatform._changeVoteTotals` accepts a `_noWeight` argument that is never read. The "no" side is derived as the complement, `noDelta = _delta - yesDelta`, so it is fully determined by `_delta` and `_yesWeight`.

Impact

Informational. Dead parameter, gas and clarity only.

Recommendation

Remove `_noWeight` parameter.

Developer Response

Fixed in commit f0babf49a48c0d141233d5aaa721c801e962e111.

2.9.2 GaugeProposer seeds `lastEpochUsed` to `epochCount()`, delaying the first gauge round by one to two epochs

The `GaugeProposer` constructor sets `lastEpochUsed = vlCVX.epochCount()`, so the first `proposeVote()` cannot run until `epochCount()` both exceeds that value and lands on an even epoch.

Technical Details

`proposeVote()` requires `ec % 2 == 0` (even epoch) and `ec > lastEpochUsed`. With `lastEpochUsed = N` at deploy, the first eligible `ec` is the smallest even value greater than `N`: `N + 1` if `N` is odd (one epoch), `N + 2` if `N` is even (two epochs). `epochCount()` rises by one per week, so the first gauge round lands one to two weeks after deployment depending on parity. After that the cadence is the intended bi-weekly even-epoch schedule.

Impact

Informational. A one-time startup delay of one to two epochs before the first gauge proposal can be created.

Recommendation

Confirm the startup delay is acceptable. If a sooner first round is wanted, seed `lastEpochUsed` lower in the constructor (it would still be gated to the next even epoch).

Developer Response

Acknowledged. Believe this was intention to get time for UI and double checks etc.

2.9.3 Quorum is measured against raw vlCVX supply while delegated participation is lazily synced, biasing quorum slightly harder to pass

The DAO executors check quorum as `totalVotes / vlCVX.totalSupplyAtEpoch(epoch)`. The denominator is the full raw vlCVX supply, but the numerator's delegated portion is read from the lazily-synced `Delegation` tables, so `Σ(countable voting powers)` can be slightly **less** than `vlCVX.totalSupplyAtEpoch`. A proposal can therefore fall just short of quorum even when its real participating power would clear it.

Technical Details

Both `CurveVoteExecutor.executeDaoVote` (L60) and `ResupplyVoteExecutor.executeDaoVote` (L67) evaluate `totalVotes * WEIGHT_BPS / totalSupplyAtEpoch(epoch) >= quorumBps`. The two sides have different freshness: `totalSupplyAtEpoch` reads vlCVX directly (full, current supply for the epoch), while `totalVotes` mixes fresh direct-voter weight (`vlCVX.balanceAtEpochOf`) with each delegate's **lazily-synced** delegated sum (`Delegation.balanceAtEpochOf`).
If a delegator increases its proposal-epoch power without syncing, e.g. a **relock into the current/proposal epoch**, especially after the delegate has already voted and then abstains, that increase lands in the supply denominator but never in `totalVotes` (the delegate's vote keeps the pre-relock weight, and `pendingWeightAdjustment` only applies on the delegate's next vote). So `Σ countable powers < vlCVX.totalSupplyAtEpoch`, and the quorum bar is effectively higher than intended. Only relocks matter here: a fresh `lock()` anchors the *next* epoch and is not counted in `totalSupplyAtEpoch(E)` on either side.
The direction is conservative (deflation only): quorum can become harder to meet, never easier. The gap is bounded to unsynced delegated-and-abstaining relock increases, and it self-heals:

anyone can call the permissionless `Delegation.syncAtEpoch(delegator, epoch)` (which reads vlCVX directly, no staleness) before finalization to restore the count, and an attacker cannot force the undercount (you cannot de-sync a user).

Impact

Informational. Liveness/fairness only: a borderline proposal supported via unsynced delegations could finalize just below quorum and fail to execute. No security impact, it cannot push a proposal over quorum that shouldn't pass, cannot be attacker-forced, and is correctable by a permissionless sync.

Recommendation

Sync delegators before finalization (a keeper sweep of `syncAtEpoch` over the delegators of voting delegates), and/or document that delegated power must be synced to count toward quorum. A `Delegation` maintained synced-supply total used as the denominator would make numerator and denominator freshness-consistent, but might not be worth implementing.

Developer Response

Acknowledged.

2.9.4 FxGaugeRegistry.setGauge is permissionless, contradicting the documented owner-curated model

`FxGaugeRegistry.setGauge` can be called by anyone, while the README describes an owner-curated F(x) gauge list.

Technical Details

`setGauge` is not `onlyOwner`; it only requires the gauge to pass the F(x) controller's validity/active check (a non-gauge reverts in `gauge_types`, so fake gauges are rejected). Anyone can therefore register any *valid* F(x) gauge as eligible. This matches the permissionless Curve registry model but diverges from the README's owner-curated F(x) description. The owner retains the `forceRemove` veto.

Impact

Informational. Eligibility-only (holders still choose what to vote on); no fake-gauge path. Code/documentation mismatch.

Recommendation

Add `onlyOwner` to `setGauge` if curation is intended, or update the README to reflect the permissionless eligibility model.

Developer Response

Readme updated at 2b18d2e2d5891efe6c463516912fc3aa3e871b71

2.9.5 DAO executors submit a full "no" vote upstream for a zero-participation proposal when quorum is disabled

`CurveVoteExecutor.executeDaoVote` and `ResupplyVoteExecutor.executeDaoVote` compute `nay` as the complement of `yay`, so a proposal with zero votes executes as `yay = 0, nay = 10000`, a full "no" upstream rather than an abstain.

Technical Details

`yay = totalVotes > 0 ? yesVotes * WEIGHT_BPS / totalVotes : 0` then `nay = WEIGHT_BPS - yay`. With `quorumBps == 0` the quorum check is skipped, so a finalized zero-vote proposal reaches the external call with `(0, 10000)`. With a positive quorum it reverts `QuorumNotMet` instead. The deploy script uses a non-zero default quorum (1500 bps), so this requires an explicit zero-quorum configuration.

Impact

Informational. Under zero quorum, a no-participation proposal is forwarded as unanimous opposition rather than abstention, misrepresenting vlCVX-holder intent on the upstream protocol.

Recommendation

Skip the upstream submission entirely (or treat as abstain) when `totalVotes == 0`, or document that zero participation is intentionally cast as a full "no".

Developer Response

Acknowledged. Could view as misrepresenting or could view as a safety fallback for no participation. Either way No votes are safer than yes votes, so I think its fine this way.

2.9.6 A finalized gauge round that misses its epoch week is stranded entirely

`CurveGaugeExecutor.executeGaugeVote` (and `FxGaugeExecutor.executeGaugeVote`) reverts `EpochExpired` once the proposal's epoch week has passed. A round whose execution slips past the Thursday boundary can therefore never be submitted, and the bi-weekly slot's emissions fall back to the previous round's weights for **both** weeks.

Technical Details

Both the Curve and F(x) gauge controllers are weekly Curve-style forks (the F(x) one is "an almost-identical fork of Curve's GaugeController"): `vote_for_gauge_weights` schedules the

new weight at `next_time = (block.timestamp + WEEK) / WEEK * WEEK` (the next Thursday boundary) and enforces a 10-day per-`(voter, gauge)` cooldown (`WEIGHT_VOTE_DELAY = 10 days`). Gauge proposals are bi-weekly, so each round governs two emission weeks. For a round on epoch week X:

- **On time** (executed within week X): vote effective the next Thursday → covers emission weeks X+1 **and** X+2.
- **Stranded** (`EpochExpired`, executed after the week-X boundary): never submitted → both X+1 and X+2 keep the **previous** round's weights (0 of 2 weeks reflect the new vote).
- **~1-2 days late** (if allowed): vote effective the following Thursday → week X+1 keeps old weights, week **X+2 gets the new allocation** (1 of 2 weeks, strictly better than stranding).

The 10-day cooldown still holds for the next round as long as its execution is spaced 10 days after the late one, which the 14-day bi-weekly cadence and 7-day window accommodate.

Impact

Informational. Liveness: a keeper that misses the epoch-week boundary forfeits the entire round's allocation (both emission weeks revert to the prior vote). A small grace would recover one of the two weeks. No funds at risk.

Recommendation

Relax the `EpochExpired` check in both executors by a small grace (about 1-2 days) past the epoch-week boundary, so a late round still lands its second emission week instead of stranding entirely. Document that after a grace-late execution the keeper must space the next round's submission 10 days later to respect `WEIGHT_VOTE_DELAY` (10 days on both Curve and F(x)).

Developer Response

Acknowledged. Will keep the hard rule. Either way it would be a loss for anyone doing vote incentives if the votes came in on the next epoch. So no reason to promote being slow.

2.9.7 Permissionless caller can redirect the gauge-vote rounding dust to a gauge of their choice

Both `CurveGaugeExecutor.executeGaugeVote` and the identical `FxGaugeExecutor.executeGaugeVote` are permissionless. Each gauge's basis-point weight is `gt * WEIGHT_BPS / totalVotes`, floored by integer division, so the submitted weights sum to slightly under 10000. The executor recovers the lost residual by padding a single gauge:

```
if (count > 0 && newCount >= totalGaugeCount && newWeight < WEIGHT_BPS) {
    weights[lastNonZero] += WEIGHT_BPS - newWeight;
}
```

`lastNonZero` is simply the last index in the caller-supplied `gauges` array whose rounded weight is nonzero. Since the caller supplies and orders that array, the caller decides which gauge becomes `lastNonZero` and so receives the entire rounding residual on the upstream gauge controller.

Impact

Informational. The residual is bounded: each gauge loses under 1 bps to flooring, so across N voted gauges it sums to under roughly N bps (tens of bps for a large round). It is small, but a permissionless caller can steer that dust to any voted gauge of their choice, so the upstream allocation differs from a neutral distribution.
Related edge: `lastNonZero` defaults to 0 and is only set when `weights[i] > 0`. If a (final) batch holds only gauges that round to zero weight, `lastNonZero` stays 0 and the residual is dumped onto `weights[0]`, a gauge that earned essentially nothing.

Recommendation

Make the pad target deterministic instead of caller-controlled: pad the largest-weight gauge, distribute the residual proportionally, or gate `executeGaugeVote` to a trusted keeper. Initialise `lastNonZero` to a sentinel so the pad can never land on a zero-weight gauge.

Developer Response

Decided we are ok with the dust going anywhere. whoever calls gets to decide. didnt want to complicate logic just for dust. however i did do one change in that lastNonZero defaulted to 0 so the last batch could point to a gauge that didnt even have any votes. so now at least making sure lastNonZero was set, and then give the dust to that gauge.
commit: 35145574bee1682713a2cc8792adabe520866f62

2.9.8 Zero-vote gauge rounds cannot submit the zero-weight revokes required to clear prior external gauge allocations

Curve's `GaugeController` persists per-gauge vote state and aggregate used power until the same gauge is voted again, including with zero weight. F(x) forwards the same `vote_for_gauge_weights(address,uint256)` call shape to its live gauge controller, which likewise maintains persistent controller-side vote state for the Convex proxy. In both integrations, omitting a gauge from the next local round does not clear its prior external allocation; that allocation must be explicitly overwritten with a zero-weight vote. Both local gauge executors, however, divide every caller-supplied gauge by `votePlatform.voteTotals(proposalId)` before forwarding it. A finalized zero-vote round has `voteTotals == 0`, so the required zero-weight revoke path is unreachable while `isDone` still reports true.

Technical Details

Curve upstream state makes explicit zero-weight overwrite necessary.
`GaugeController.vote_for_gauge_weights` reads the old per-gauge slope, recomputes aggregate `vote_user_power`, and only replaces a gauge's old state when that same gauge is voted again:

```
# integrated_code/curve/contracts/GaugeController.vy
old_slope: VotedSlope = self.vote_user_slopes[msg.sender][_gauge_addr]
new_slope: VotedSlope = VotedSlope({
    slope: slope * _user_weight / 10000,
    end: lock_end,
```

```
    power: _user_weight
})

power_used: uint256 = self.vote_user_power[msg.sender]
power_used = power_used + new_slope.power - old_slope.power
self.vote_user_power[msg.sender] = power_used
assert (power_used >= 0) and (power_used <= 10000), 'Used too much power'

self.vote_user_slopes[msg.sender][_gauge_addr] = new_slope
```

If old gauge A is omitted from a later submission, its prior `old_slope.power` is never replaced and remains part of the controller's live state. A zero-weight call to the same gauge is therefore the revoke operation.
F(x) uses the same downstream voting primitive. The Booster forwards each gauge entry as `vote_for_gauge_weights(address,uint256)`:

```
// integrated_code/fx/contracts/contracts/Booster.sol
function voteGaugeWeight(address _controller, address[] calldata _gauge, uint256[]
calldata _weight) external onlyOwner {
    for(uint256 i = 0; i < _gauge.length; ) {
        bytes memory data = abi.encodeWithSelector(
            bytes4(keccak256("vote_for_gauge_weights(address,uint256)")),
            _gauge[i],
            _weight[i]
        );
        _proxyCall(_controller, data);
        unchecked { ++i; }
    }
}
```

F(x)'s reward calculation also derives downstream emissions from controller-side `gauge_relative_weight(_gauge)`, showing that these votes are persistent controller state with real reward consequences:

```
// integrated_code/fx/contracts/contracts/boosting/PoolUtilities.sol
uint256 emissionRate = IFxnToken(fxn).rate() * 1e18;
uint256 gaugeWeight = IGaugeController(gaugeController).gauge_relative_weight(_gauge);
```

Live `eth_call` against the fixed F(x) controller `0xe60e...` shows that the Convex proxy address currently has `vote_user_power == 10000`, confirming that its prior gauge allocation remains resident in controller state outside any local proposal lifecycle and must be explicitly reallocated or revoked rather than automatically cleared per round.
By contrast, Resupply's upstream `Voter` records votes per `(account, proposalId)` and has no cross-round per-target allocation state. Each proposal is independent:

```
// integrated_code/resupply/src/dao/Voter.sol
Vote memory vote = accountVoteWeights[account][id];
require(vote.weightYes + vote.weightNo == 0, "Already voted");
...
accountVoteWeights[account][id] = vote;
```

This explicit revoke requirement is therefore specific to the gauge integrations, not a generic assumption imported from local docs.
The local executors still compute every supplied gauge weight before distinguishing current positive entries from zero-weight revokes:

```
// src/CurveGaugeExecutor.sol and src/FxGaugeExecutor.sol
uint256 totalVotes = votePlatform.voteTotals(proposalId);
...
uint256 gt = votePlatform.gaugeTotal(proposalId, gauges[i]);
weights[i] = gt * WEIGHT_BPS / totalVotes;
```

For zero-vote finalized rounds, the platform has no current positive gauge entries:

```
// src/GaugeVotePlatform.sol
function getGaugeCount(uint256 _proposalId) external view returns (uint256) {
    return _gaugeEntries[_proposalId].length;
}
```

`isDone` checks only equality between submitted count and current positive gauge count:

```
function isDone(uint256 proposalId) external view returns (bool) {
    if (!votePlatform.isFinalized(proposalId)) return false;
    return _executionState[proposalId].gaugeCount == votePlatform.getGaugeCount(
proposalId);
}
```

With no votes, both values are zero, so the round appears done before any zero-weight revoke can be submitted. The controller state that actually needs to be cleared still exists upstream.

Impact

If a scheduled Curve or F(x) gauge round receives no votes, prior external gauge allocations do not automatically disappear upstream; they remain live until explicitly overwritten. The current zero-vote executor path makes that overwrite impossible, so stale external allocations persist while `isDone()` incorrectly reports completion. On Curve, those retained allocations can block later reallocations with `Used too much power`. On F(x), the same persistent controller state means prior allocations remain active until a later authorized overwrite or out-of-band intervention. In combination with the separate missing previous-round cleanup tracking issue, funded rounds require off-chain discovery and ordering of revokes, while zero-vote rounds cannot submit revokes at all.

Recommendation

Handle `totalVotes == 0` explicitly. For zero-vote proposals, allow caller-supplied revoke gauges to be forwarded with zero weights and track revoke completion separately from positive current-proposal gauge completion. `isDone` should not report true for a zero-vote round while prior external allocations that require explicit zero-weight overwrite remain unresolved.

Developer Response

Acknowledged. Dont think this is an issue to me. if no one votes then should just keep the previous weight.

2.9.9 F(x) gauge executor is registered without the required ConvexCore operator grant

`FxGaugeExecutor` now submits F(x) gauge weights through `ConvexCore.execute()`, so the executor contract itself must be present in `ConvexCore.operators`. The deployment script deploys and registers the executor as the live F(x) gauge executor, but it does not call `ConvexCore.setOperator(address(fxGaugeExecutor), true)`. As a result, a finalized F(x) gauge round can pass all local voting checks and still revert with `"Not operator"` before any upstream gauge-voting call is attempted.

Technical Details

The current executor resolves the active F(x) operator through the voter proxy and forwards the vote through `ConvexCore`:

```
// src/FxGaugeExecutor.sol
address booster = IFxVoterProxy(fxVoterProxy).operator();
convexCore.execute(
    booster,
    abi.encodeWithSelector(IFxGaugeVoter.voteGaugeWeight.selector, gaugeController, gauges, weights)
);
```

`ConvexCore.execute()` only accepts calls from addresses already marked as operators:

```
// src/ConvexCore.sol
modifier onlyOperator() {
    require(operators[msg.sender], "Not operator");
    _;
}

function execute(address target, bytes calldata data) external onlyOperator returns (bytes memory) {
    (bool success, bytes memory returnData) = target.call(data);
    ...
}

function setOperator(address _operator, bool _active) external onlyOperator {
    ...
}
```

The deployment script registers the executor but does not itself grant this role:

```
// script/Deploy.s.sol
FxGaugeExecutor fxGaugeExecutor = new FxGaugeExecutor(
    "Fx Gauge Executor",
    address(fxGaugeVoting),
    address(core)
);

core.execute(
    address(registry),
    abi.encodeWithSignature("setVotingContract(string,uint8,address)", "FX", GAUGE_EXECUTOR, address(fxGaugeExecutor))
);
```

If the operator grant is omitted, settlement reverts with `"Not operator"` until an existing `ConvexCore` operator adds the executor.

Impact

If deployment or migration registers `FxGaugeExecutor` without the matching `ConvexCore` operator grant, finalized F(x) gauge votes cannot be settled through the intended path until an existing `ConvexCore` operator grants that role.

Recommendation

Authorize `FxGaugeExecutor` in `ConvexCore.operators` before relying on it as the live F(x) gauge executor. An existing `ConvexCore` operator should call

`setOperator(address(fxGaugeExecutor), true)` or the equivalent self-call through `ConvexCore.execute()`, and deployment checks should verify that authorization.

Developer Response

Acknowledged. not going to hook up the executors immediately when we deploy (phased deployment). and its not something the deployer can do either for production. so not adding to the deploy script

2.9.10 Curve executor operator assignment is an owner-controlled VoteDelegateExtension setup step

`CurveVoteExecutor` and `CurveGaugeExecutor` settle through a fixed external `VoteDelegateExtension`. That extension accepts DAO and gauge submissions only from its configured `daoOperator` and `gaugeOperator`, and those roles are set by the extension owner. The local deployment script registers fresh executors in `VotingRegistry`, but the corresponding upstream owner transaction is a separate activation step. Under a trusted-owner deployment model, this is better treated as an operator-runbook prerequisite than a standalone Medium-severity vulnerability.

Technical Details

The local executors call the external extension directly:

```
// src/CurveVoteExecutor.sol
voteDelegate.DaoVoteWithWeights(proposalId, yay, nay, isOwnership);

// src/CurveGaugeExecutor.sol
voteDelegate.GaugeVote(gauges, weights);
```

The extension itself keeps distinct DAO and gauge operator slots under owner control:

```
// integrated_code/convex-platform/contracts/contracts/VoteDelegateExtension.sol
modifier onlyDaoOperator() {
    require(daoOperator == msg.sender, "!dop");
    _;
}

modifier onlyGaugeOperator() {
    require(gaugeOperator == msg.sender, "!gop");
    _;
}

function setDaoOperator(address _operator) external onlyOwner {
    daoOperator = _operator;
}

function setGaugeOperator(address _operator) external onlyOwner {
    gaugeOperator = _operator;
}
```

The deployment script creates and registers the executors locally, but it does not itself complete the external owner-side role update on `VoteDelegateExtension`. Live mainnet reads also show that `owner`, `daoOperator`, and `gaugeOperator` currently remain on the multisig, confirming that this is an explicitly owner-controlled configuration surface rather than an uncontrolled permission gap.

Impact

If a deployment or migration activates new Curve executors without the matching `setDaoOperator` / `setGaugeOperator` owner transaction, Curve settlement remains unavailable until the owner completes that configuration. In a trusted-owner model, this is an operational activation dependency rather than a direct exploit path.

Recommendation

Document `VoteDelegateExtension` operator assignment as an explicit owner-runbook step. When rotating or deploying Curve executors, the extension owner should set `daoOperator` to the intended DAO executor and `gaugeOperator` to the intended gauge executor or approved adapter, and activation should verify those roles before relying on the executors.

Developer Response

Acknowledged. not going to hook up the executors immediately when we deploy (phased deployment). and its not something the deployer can do either for production. so not adding to the deploy script

2.9.11 Owner-seeded self-delegation can double-count a holder's weight

The user-controlled `setDelegate` path rejects self-delegation, but the owner-only bootstrap `seedDelegates` path does not enforce the same invariant. If a self-delegation is seeded before initialization is closed, the voting platforms can treat the user's weight as both base vlCVX weight and delegated-in weight.

Technical Details

Bootstrap seeding accepts any nonzero delegate:

```
// src/Delegation.sol
function seedDelegates(address[] calldata _users, address[] calldata _delegates)
external onlyOwner {
    ...
    address user = _users[i];
    address delegate = _delegates[i];
    if (delegate == address(0)) revert NoDelegate();
    ...
    history.push(SetDelegateRecord({delegate: delegate, startingEpoch: uint32(nextEpoch)
}));
    _syncUser(user, delegate, nextEpoch, FILL_EPOCHS);
}
```

The normal user path rejects the same relation:

```
function setDelegate(address _delegate) external {
    if (_delegate == msg.sender) revert SelfDelegation();
    ...
}
```

Voting initialization adds raw base vlCVX weight and delegated-in weight separately:

```
uint256 baseWeight = vlCVX.balanceAtEpochOf(epoch, _account);
uint256 totalDelWeight = delegation.balanceAtEpochOf(epoch, _account);

user.baseWeight = uint96(baseWeight);
user.adjustedWeight += int96(int256(totalDelWeight));
```

If Alice is seeded as her own delegate, the same lock can appear in both components.

Impact

A bad bootstrap seed can inflate a user's voting influence in local DAO and gauge tallies. The action is owner-only and bootstrap-only, so this is informational under a trusted-owner deployment model.

Recommendation

Apply the same `delegate == user` rejection in `seedDelegates` that `setDelegate` already uses.

Developer Response

Fixed in commit 004f60da381929c7900bc785e0907427fa5cca75.

2.9.12 Resupply executor requires upstream PermaStaker operator setup

`ResupplyVoteExecutor` relays results through the fixed Resupply `PermaStaker.safeExecute` path, and that upstream contract accepts calls only from its owner or configured operator. The repository deployment script registers a fresh `ResupplyVoteExecutor`, but the required `PermaStaker` operator assignment is controlled by the upstream owner and is not performed from this repo. Under a tightly coordinated Convex/Resupply deployment model, this is better treated as a deployment prerequisite than a standalone Medium-severity vulnerability.

Technical Details

The executor assumes the upstream staker will already trust the calling contract:

```
// src/ResupplyVoteExecutor.sol
bytes memory voterCall = abi.encodeWithSelector(
    IResupplyVoter.voteForProposal.selector,
    RESUPPLY_STAKER,
    proposalId,
    yay,
    nay
);

IResupplyStaker(RESUPPLY_STAKER).safeExecute(RESUPPLY_VOTER, voterCall);
```

That call succeeds only if `msg.sender` is the `PermaStaker` owner or its configured operator:

```
// integrated_code/resupply/src/dao/tge/PermaStaker.sol
modifier onlyOwnerOrOperator {
    require(msg.sender == owner() || msg.sender == operator, "!ownerOrOperator");
    _;
}
```

```
function safeExecute(address target, bytes calldata data) external returns (bytes memory
) {
    (bool success, bytes memory result) = _execute(target, data);
    require(success, "CallFailed");
    return result;
}

function setOperator(address _operator) external onlyOwner {
    operator = _operator;
}
```

The local deployment script deploys and registers `ResupplyVoteExecutor`, but it cannot complete that upstream trust assignment on its own. The integration therefore relies on a separate operator-setup step performed by the trusted `PermaStaker` owner or by an aligned upstream runbook.

Impact

If the upstream operator assignment is omitted during deployment or activation, Resupply vote execution reverts until that setup step is completed. In a trusted, closely coordinated deployment model, this is an operational readiness dependency rather than an independent permission-bypass bug.

Recommendation

Document `PermaStaker` operator assignment as an explicit activation step. Before enabling Resupply execution, the trusted upstream owner should set the deployed `ResupplyVoteExecutor` or an approved adapter as operator and verify that authorization out of band.

Developer Response

Acknowledged. not going to hook up the executors immediately when we deploy (phased deployment). and its not something the deployer can do either for production. so not adding to the deploy script

Convex - Onchain Voting

Completed 2026-06-11