Aragon DAO Gov Plugin Security Review

Auditors

Emanuelle Ricci, Lead Security Researcher

Om Parikh, Security Researcher

Patrick Drotleff, Security Researcher

Report prepared by: Lucas Goiriz

August 28, 2025

Contents

1 About Spearbit

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

Learn more about us at spearbit.com

2 Introduction

Aragon gives organizations the tools to build, govern, and accrue value effectively onchain.

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

3 Risk classification

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

3.1 Impact

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

3.2 Likelihood

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

3.3 Action required for severity levels

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

4 Executive Summary

Over the course of 9 days in total, Aragon engaged with Spearbit to review the lock-to-vote-plugin protocol. In this period of time a total of **38** issues were found.

Summary

Project Name	Aragon
Repository	lock-to-vote-plugin
Commit	b3c5503c
Type of Project	Infrastructure, Governance
Audit Timeline	Jul 28th to Aug 6th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	6	6	0
Low Risk	12	7	5
Gas Optimizations	1	1	0
Informational	17	9	8
Total	**38**	**25**	**13**

5 Findings

5.1 High Risk

5.1.1 `MinVotingPowerCondition` logic can be bypassed via flashloans

Severity: High Risk

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

Description: The `MinVotingPowerCondition` contract allows any users who hold at least `plugin.minProposerVotingPower()` amount of tokens (locked in the `LockManager` or in their token balance) to create a proposal and execute `LockToVote.createProposal`. This logic can be easily bypassed if the `token` allow flashloan or flashmint. Given that `isGranted` checks the user's token balance the user does not even need to lock the flashloaned amount into the `LockManager` to be able to create and spam proposals.

The flashloan issue is even more problematic when combined with the voting mode `EarlyExecution` and the fact that proposals can (in that mode) be created and executed in the same block/timestamp. A user could simply:

1. Flashloan the token.
2. Create the proposal.
3. Lock + vote the proposal and auto-execute it (`EarlyExecution`).
4. Unlock the tokens.
5. Repay the flashloan.

All in the very same transaction.

Recommendation: These are some ideas that could help the Aragon team to brainstorm a possible solution:

- Avoid using the token's balance and require the user to only use the locked amount.
- Store the proposal's creator in the proposal struct and require that the locked amount required to create the proposal won't be unlockable until the proposal ends. Keep in mind that with voting mode `EarlyExecution`, users would be able to anyway exploit this requirement via flashloan.
- Avoid executing the transaction in the same block that the "deciding" vote has been made. This would require storing and tracking an additional flag in the proposal (at least for the `EarlyExecution` case). If the vote would have allowed an "early execution", that flag would be turned to `true` and any executor would be able to execute the proposal from the next block (since the vote). This new flag would need to be integrated across all the other functions that check the state of the proposal (open, succeeded, executed, can be voted, and so on).

> Note that these are suggestions that could lead to side effects and need to be further properly evaluated from a security standpoint.

Aragon: Fixed in PR 28.

Spearbit: Fix verified.

5.1.2 Proposals created with voting mode `EarlyExecution` are vulnerable to flashloan attacks

Severity: High Risk

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

Description: If the token used by the `LockManager` can be flashloaned (or flashminted) and a proposal is created with the voting mode `EarlyExecution`, anyone could be able to "early execute" it performing the following action.

- Flashloan the needed amount.
- Lock the flashloaned amount.

- Cast a "YES" vote and trigger the "early execute" logic.
- Unlock the tokens.
- Repay the flashloan.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.17;

import {TestBase} from "./lib/TestBase.sol";
import {DaoBuilder} from "./builders/DaoBuilder.sol";
import {DAO} from "@aragon/osx/src/core/dao/DAO.sol";
import {Action} from "@aragon/osx-commons-contracts/src/executors/IExecutor.sol";
import {LockToVotePlugin, MajorityVotingBase} from "../src/LockToVotePlugin.sol";
import {IMajorityVoting} from "../src/interfaces/IMajorityVoting.sol";
import {LockManagerERC20} from "../src/LockManagerERC20.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {TestToken} from "./mocks/TestToken.sol";

import "forge-std/console.sol";

contract SVoteFlashloanTest is TestBase {
    DaoBuilder builder;
    DAO dao;
    LockToVotePlugin ltvPlugin;
    LockManagerERC20 lockManager;
    IERC20 lockableToken;
    uint256 proposalId;

    // Default actions for proposal creation
    Action[] internal actions;

    event ProposalExecuted(uint256 indexed proposalId);

    function setUp() public {}

    function _init(MajorityVotingBase.VotingMode _votingMode) internal {
        builder = new DaoBuilder();
        (dao, ltvPlugin, lockManager, lockableToken) =
        ↪  builder.withVotingPlugin().withVotingMode(_votingMode).build();
    }

    function testFlashLoanVote() public {
        _init(MajorityVotingBase.VotingMode.EarlyExecution);

        // give alice the proposer permission to make it simple
        dao.grant(address(ltvPlugin), alice, ltvPlugin.CREATE_PROPOSAL_PERMISSION_ID());

        // NOTE: I need to provide the permission to the LockManager instead of Alice because
        // `vote` is passing `msg.sender` (the lock manager) to `_attemptEarlyExecution` instead of the
        ↪  actual voter address
        dao.grant(address(ltvPlugin), address(lockManager), ltvPlugin.EXECUTE_PROPOSAL_PERMISSION_ID());

        vm.prank(alice);
        proposalId = ltvPlugin.createProposal("0x", new Action[](0), 0, 0, abi.encode(uint256(0)));

        // flashloan to alice just enough token to make the proposal pass and execute it
        _flashloan(address(lockableToken), alice, 10e18);

        // approve the token to be locked
        vm.prank(alice);
```

```
        lockableToken.approve(address(lockManager), 10e18);

        // lock and vote the Yes
        // check that the proposal has been executed by expeting the `ProposalExecuted` event
        vm.expectEmit();
        emit ProposalExecuted(proposalId);
        vm.prank(alice);
        lockManager.lockAndVote(proposalId, IMajorityVoting.VoteOption.Yes, 10e18);

        // alice has not more token, everthing has been temporary locked in the lockmanager
        assertEq(lockableToken.balanceOf(alice), 0);

        // alice can unlock and withdraw everything because the proposal has been auto-executed by her
        ↪  vote
        vm.prank(alice);
        lockManager.unlock();

        // alice got all her 10e18 token back and can repay the 1000 token flashloan
        assertEq(lockableToken.balanceOf(alice), 10e18);
    }

    function _flashloan(address token, address user, uint256 amount) internal {
        TestToken(token).mint(user, amount);
    }
}
```

Recommendation: One possible path to be further evaluated is to avoid allowing the early execution in the very same block that the vote has been made. This would require tracking the "success" of a proposal in a separate flag, stored in the proposal struct.

If the flag is `true` (or the the voting period has ended and the existing success logic passes) the proposal can be executed. This new flag and block requirement needs to be integrated across the existing logic in all the functions like `canVote`, `canExecute`, `hasSucceeded`, `_isProposalOpen` and so on.

> Note that these are suggestions that could could lead to side effects and need to be further properly evaluated from a security standpoint.

Aragon: Fixed in PR 26.

Spearbit: The issue has been resolved with the PR 26

As previously mentioned, by removing the `EarlyExecution` voting mode, users won't be able to "withdraw" their vote and unlock them before all the proposals they have voted in have ended (voting period has ended).

This is not a "security issue" because it follows the intended behavior of the "Standard Voting Mode", but it could create a UX issue because no one will be willing to vote on multiple proposals or will "wait" until the very end of the proposal voting period to lock their token only for the smallest amount of time (just to make a couple of examples).

Unless tokens are valueless (can't be traded), users could want to avoid locking their token and losing money. This will lead, as explained above, to voting behaviors that will make the proposal experience quite worse.

Aragon: Agree with your feedback. However, the early execution mode was something we inherited from our existing plugins and in this context we see it more as a risk than as an asset.

Even if the UX is impacted, and even if there could be a way to execute on the next block, we prefer being extra safe here. As mentioned, the likelihood of early execution is very, very low from our experience. So the impact of removing this mode should be close to nothing.

- We are removing the lingering appearences of "early" that weren't removed yet.
- We are going ahead with the PR that you verified.
- We acknowledge the feedback regarding the UX degratation.

5.2 Medium Risk

5.2.1 Lack of `SafeERC20` can inflate user balance

Severity: Medium Risk

Context: LockManagerERC20.sol#L35-L43

Summary: If the `erc20Token` configured within `LockManagerERC20` is a token that does not revert due to insufficient approved funds, the `msg.sender` can inflate their `lockedBalances[msg.sender]` by an arbitrary `_amount` without actually sending any token.

Finding Description: There are some ERC20 token that do not `revert()` when a transfer fails, eg. due to a lack of funds or approval to make use of a user's funds. Instead, they return `0` or `false` which `_doLockTransfer` is currently not checking. As visible by the inline comment, it explicitly assumes that the `transferFrom()` call would revert _if not enough balance is approved.

Flattened code snippet based on `LockManagerBase` and `LockManagerERC20`:

```
abstract contract LockManagerBase is ILockManager {

  function lock(uint256 _amount) public virtual {
      if (_amount == 0) revert NoBalance();

      /// @dev Reverts if not enough balance is approved
      erc20Token.transferFrom(msg.sender, address(this), _amount);

      lockedBalances[msg.sender] += _amount;
      emit BalanceLocked(msg.sender, _amount);
  }
```

Impact Explanation: An attacker can specify an arbitrary `_amount` without having ever given any approval of his `erc20Token` to the `LockManagerERC20`. The attackers balance `lockedBalances[msg.sender]` still increases allowing him to:

- Gain unlimited voting power.
- Drain the `LockManagerERC20` of its funds deposited by other users.

Likelihood Explanation: This is relatively unlikely to happen since such tokens are rare. Some examples include: Basic Attention Token (BAT), Huobi Token (HT), Compound USD Coin (cUSDC), 0x Protocol Token (ZRX).

Recommendation: Use `SafeERC20` across all contracts that interact with an ERC20 token.

```
+ using SafeERC20 for IERC20;

  /// @inheritdoc LockManagerBase
  function _doLockTransfer(uint256 _amount) internal virtual override {
-       erc20Token.transferFrom(msg.sender, address(this), _amount);
+       erc20Token.safeTransferFrom(msg.sender, address(this), _amount);
  }

  /// @inheritdoc LockManagerBase
  function _doUnlockTransfer(address _recipient, uint256 _amount) internal virtual override {
-       erc20Token.transfer(_recipient, _amount);
+       erc20Token.safeTransfer(_recipient, _amount);
  }
```

Aragon: Fixed in PR 32.

Spearbit: Fix verified.

5.2.2 Lock fails for unlimited approvals

Severity: Medium Risk

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

Summary: The `lock()` function intends to lock the user's entire approved balance, but this will fail for the most common case: Most users give the contract's they interact with an unlimited allowance by setting it to `type(uint256).max`. The `lock()` function attempts to transfer this amount to itself, which is bound to fail.

Finding Description: The `LockManagerBase` contract has two `lock()` functions, both intended to lock a balance of tokens into the contract. One variant of these functions does not require the specification of a specific amount, instead, it assumes that the user wants to lock all of the tokens that have been approved to the contract according to `_incomingTokenBalance()`.

The `LockManagerERC20` contract implements `_incomingTokenBalance()` by simply returning the current allowance given by the user. But for ERC20 token the most commonly given allowance is the "unlimited" one, represented by setting it to `type(uint256).max`. This doesn't accurately represent the actual incoming token balance and will cause the `transferFrom()` call to fail.

Recommendation: Have the `_incomingTokenBalance()` return either the actual balance or the allowance depending on whichever is smallest.

```
  function _incomingTokenBalance() internal view virtual override returns (uint256) {
-     return erc20Token.allowance(msg.sender, address(this));
+     uint256 allowance = erc20Token.allowance(msg.sender, address(this));
+     uint256 balance = erc20Token.balanceOf(msg.sender);
+     return (allowance >= balance) ? balance : allowance;
  }
```

Aragon: Fixed in PR 31.

Spearbit: Fix verified.

5.2.3 Misuse of `isProposalOpen()` function

Severity: Medium Risk

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

Summary: The `isProposalOpen()` function returns true if it's open for voting. It returns false, if it's too late *or* too early for voting. But in two places of the codebase it seems to be used as a `isProposalStillOpen()` function, forgetting or ignoring the fact that it will also be false when it hadn't even been voted on yet, leading to unintended behavior.

Description:

- Occurrence 1: `LockManagerBase._withdrawActiveVotingPower()`: The `_withdrawActiveVotingPower()` function iterates through the list of `knownProposalIds` and checks for each *whether (1) the proposal is still open with `isProposalOpen()`* (if not, its identifier is simply removed from the list) and whether (2) the user has active voting power on the proposal (if yes, it attempts clearing the vote). The problem here lies in handling proposals that haven't even started yet: The function would remove them from the `knownProposalIds` list, and once it actually starts, it wouldn't check it anymore. This would allow a user to vote with their tokens and then withdraw them without having their vote cleared.

  Disclaimer: This issue was identified by Aragon themselves early during the audit.

- Occurrence 2: `MajorityVotingBase._hasSucceeded()`: The `_hasSucceeded()` function checks whether a proposal has been successfully voted for, such that it can be executed. It checks *whether the proposal is still open with `isProposalOpen()`* to decide whether to apply standard or early execution threshold checks. But a proposal that hasn't even started yet would end up using standard threshold checks instead of the correct early execution checks. Fortunately, this has no impact thanks to the fact that a proposal needs at least 1 wei of voting power to pass thresholds, even if those have been set to 0, and a proposal that hasn't started yet cannot be voted on.

Recommendation: To prevent such confusions, and possible bugs, we encourage the creation and use of a `isProposalEnded()` function specifically for this use case. It should return `true` iff both the start and end dates are in the past.

Aragon: Fixed in PR 42.

Spearbit: Fix verified.

5.2.4 Proposal's action should not be able to target the voting contract or the lock manager

Severity: Medium Risk

Context: LockToVotePlugin.sol#L121-L125

Description: The `LockToVotePlugin.createProposal` contract is not performing any sanity checks on the actions bound to the proposal that will be executed by the `IExecutor` contract once the proposal has passed and is executed.

If the plugin `_targetContract` is configured to execute the execute call with a low-level `deleagatecall` (`Operation.DelegateCall`) and the action are targeting the `LockManager`, those actions will be executed "on behalf" of the `LockToVotePlugin` and will be able to trigger the `proposalCreated` and `proposalEnded` functions even if the `LcokToVotePlugin` is not "actively" invoking them because of a direct user's action.

This could allow the attacker to craft proposals that would:

- Spam the `proposalCreated` function, adding "fake" proposal IDs to the `knownProposalIds` set and possibly DDoSing the lazy garbage collector logic performed by the `_withdrawActiveVotingPower` function during the unlock operation. Users won't be able to unlock and withdraw their tokens forever.
- Invoke the `proposalEnded` even if the proposal has not been passed, executed, or even started. This would allow the attacker to cast a vote on a proposal, exploit the system by invoking `proposalEnded` and unlock (and withdraw) their token without cleaning their vote.

Let's make an example for the above scenario.

- A "normal" `proposal_1` is created. The proposal ends in 1 year.
- A "crafted" `proposal_2` is created. The actions of this proposal will invoke `LockManager.proposalEnded(proposal_1_id)`. This proposal ends in a tiny amount of time, but this is not relevant; with the existing codebase we could early-execute them or even leverage the flashloan exploit already described during the review.

1. Alice locks 100 tokens.
2. Alice casts a "NO" vote to `proposal_1`.
3. Alice casts a "YES" vote to `proposal_2`.
4. `proposal_2` passes and is executed. One of the actions will invoke `LockManager.proposalEnded(proposal_1)`, removing `proposal_1` from the list of the `knownProposalIds` set.
5. Alice can now call `LockManager.unlock` and will be able to unlock and withdraw their tokens even if in reality her vote is still registered in `proposal_1`.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.17;

import {TestBase} from "./lib/TestBase.sol";
import {DaoBuilder} from "./builders/DaoBuilder.sol";
import {DAO, IDAO} from "@aragon/osx/src/core/dao/DAO.sol";
import {Action} from "@aragon/osx-commons-contracts/src/executors/IExecutor.sol";
import {createProxyAndCall} from "../src/util/proxy.sol";
import {LockToVotePlugin, MajorityVotingBase} from "../src/LockToVotePlugin.sol";
import {LockManagerSettings, PluginMode} from "../src/interfaces/ILockManager.sol";
```

```
import {IMajorityVoting} from "../src/interfaces/IMajorityVoting.sol";
import {LockManagerERC20} from "../src/LockManagerERC20.sol";
import {DaoUnauthorized} from "@aragon/osx-commons-contracts/src/permission/auth/auth.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {TestToken} from "./mocks/TestToken.sol";
import {ILockToVote} from "../src/interfaces/ILockToVote.sol";
import {ILockToGovernBase} from "../src/interfaces/ILockToGovernBase.sol";
import {Initializable} from "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import {IPlugin} from "@aragon/osx-commons-contracts/src/plugin/IPlugin.sol";
import {IMembership} from
↪  "@aragon/osx-commons-contracts/src/plugin/extensions/membership/IMembership.sol";
import {IERC165Upgradeable} from
↪  "@openzeppelin/contracts-upgradeable/utils/introspection/ERC165Upgradeable.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {RATIO_BASE} from "@aragon/osx-commons-contracts/src/utils/math/Ratio.sol";
import {MinVotingPowerCondition} from "../src/conditions/MinVotingPowerCondition.sol";


contract Executor {
    function execute(bytes32 _callId, Action[] calldata _actions, uint256 _allowFailureMap)
        external
        returns (bytes[] memory execResults, uint256 failureMap)
    {

        execResults = new bytes[](_actions.length);
        for (uint256 i = 0; i < _actions.length;) {
            (bool success, bytes memory result) = _actions[i].to.call(_actions[i].data);

            execResults[i] = result;
            unchecked {
                ++i;
            }
        }
    }
}

contract SVoteQuickTest is TestBase {
    DaoBuilder builder;
    DAO dao;
    LockToVotePlugin ltvPlugin;
    LockManagerERC20 lockManager;
    IERC20 lockableToken;
    uint256 proposalId;

    function setUp() public {}

    function _flashloan(address token, address user, uint256 amount) internal {
        TestToken(token).mint(user, amount);
    }

    function _init(MajorityVotingBase.VotingMode _votingMode) internal {
        builder = new DaoBuilder();
        (dao, ltvPlugin, lockManager, lockableToken) =
            builder.withVotingPlugin().withVotingMode(_votingMode).withProposer(alice).build();
    }

    function _vote(address user, uint256 propId, uint256 amount, IMajorityVoting.VoteOption vote)
    ↪  internal {
        _flashloan(address(lockableToken), user, amount);
        vm.startPrank(user);
        lockableToken.approve(address(lockManager), amount);
        lockManager.lockAndVote(propId, vote, amount);
```

```
        vm.stopPrank();
    }

    function _vote(address user, uint256 propId, IMajorityVoting.VoteOption vote) internal {
        vm.startPrank(user);
        lockManager.vote(propId, vote);
        vm.stopPrank();
    }

    function _lock(address user, uint256 amount) internal {
        _flashloan(address(lockableToken), user, amount);
        vm.startPrank(user);
        lockableToken.approve(address(lockManager), amount);
        lockManager.lock(amount);
        vm.stopPrank();
    }

    function testExploitExecute() public {
        _init(MajorityVotingBase.VotingMode.EarlyExecution);

        // grant THIS contract to modify the target config
        dao.grant(address(ltvPlugin), address(this), ltvPlugin.SET_TARGET_CONFIG_PERMISSION_ID());

        // grant lockManager to execute early proposals
        dao.grant(address(ltvPlugin), address(lockManager), ltvPlugin.EXECUTE_PROPOSAL_PERMISSION_ID());

        // replace the target
        Executor ex = new Executor();
        IPlugin.TargetConfig memory targetConfig =
            IPlugin.TargetConfig({target: address(ex), operation: IPlugin.Operation.DelegateCall});
        ltvPlugin.setTargetConfig(targetConfig);

        // alice LOCK 100e18 tokens
        _lock(alice, 100e18);

        // generate the first proposal
        vm.prank(alice);
        uint256 proposalId_1 = ltvPlugin.createProposal("0x", new Action[](0), 0, 0,
        ↪  abi.encode(uint256(0)));

        // alice vote NO to the first proposal
        _vote(alice, proposalId_1, IMajorityVoting.VoteOption.No);

        // 1 active proposal
        vm.assertEq(lockManager.knownProposalIdsLength(), 1);

        // expect revert, alice has VOTED on the first proposal that is still ONGOING
        vm.prank(alice);
        vm.expectRevert();
        lockManager.unlock();

        // generate the data for the second proposal
        bytes memory action_data = abi.encodeWithSignature("proposalEnded(uint256)", proposalId_1);
        Action[] memory actions_exploit = new Action[](1);
        Action memory action_1 = Action(address(lockManager), 0, action_data);
        actions_exploit[0] = action_1;

        vm.prank(alice);
        uint256 proposalId_2 = ltvPlugin.createProposal("0x", actions_exploit, 0, 0,
        ↪  abi.encode(uint256(0)));

        assertEq(lockableToken.balanceOf(alice), 0);
```

```
        // 2 active proposal
        vm.assertEq(lockManager.knownProposalIdsLength(), 2);

        // vote second proposal
        _vote(alice, proposalId_2, IMajorityVoting.VoteOption.Yes);

        // all the proposal have been removed from the KNOWN one, even if
        // proposal 1 is ACTIVE and ONGOING
        vm.assertEq(lockManager.knownProposalIdsLength(), 0);

        (bool proposal_1_open, bool proposal_1_executed,, MajorityVotingBase.Tally memory tally,,,) =
            ltvPlugin.getProposal(proposalId_1);
        vm.assertEq(proposal_1_open, true);
        vm.assertEq(proposal_1_executed, false);
        vm.assertEq(tally.no, 100e18);

        // alice can unlock the tokens and get back 100e18 even if the proposal1 is active
        vm.prank(alice);
        lockManager.unlock();

        assertEq(lockableToken.balanceOf(alice), 100e18);
    }

    function _createProposalId(bytes32 _salt) internal view virtual returns (uint256) {
        return uint256(keccak256(abi.encode(block.chainid, block.number, address(this), _salt)));
    }
}
```

Recommendation: Aragon should prevent the creation of new proposals via `createProposal` when the action's target is `address(this)` (the `LockToVotePlugin` contract) or `lockManager` (the `LockManager` contract).

Aragon could also consider reverting when `_targetConfig` is equal to `address(this)` or `lockManager`. Such a configuration would not make sense, and we already know that the `execute` will fail with those targets because none of them implements the `IExecutor.execute` function.

```
function execute(
    bytes32 _callId,
    Action[] memory _actions,
    uint256 _allowFailureMap
) external returns (bytes[] memory, uint256);
```

Aragon: Fixed in PR 29.

Spearbit: Aragon has decided to not restrict the target of the action itself. The PR 29 has applied the following changes:

- The executor of the proposal can't be the plugin itself or the lock manager.
- The execution of the proposal can't be configured to use a `delegatecall` to execute the proposal.

Given that neither the `LockToVotePlugin` nor the `LockManager` exposes any function with the signature `function execute(bytes32 _callId, Action[] memory _actions, uint256 _allowFailureMap)` we can consider the changes enough to solve the issue.

Aragon should keep this consideration in mind when they plan any future upgrade or change in the codebase.

5.2.5 Using idle balance in computing `isGranted` doesn't account for freezed & blacklisted tokens

Severity: Medium Risk

Context: MinVotingPowerCondition.sol#L45

Description: In `MinVotingPowerCondition.isGranted`.

```
function isGranted(address _where, address _who, bytes32 _permissionId, bytes calldata _data)
    public
    view
    override
    returns (bool)
{
    (_where, _data, _permissionId);

    uint256 _currentBalance = token.balanceOf(_who) + lockManager.getLockedBalance(_who);
    uint256 _minProposerVotingPower = plugin.minProposerVotingPower();

    return _currentBalance >= _minProposerVotingPower;
}
```

If an address holds tokens but can't spend it due being freezed or blacklisted, then they are granted access (i.e in this case, they can create proposal). However, the intended behavior should be to not allow calling create proposal as tokens are not transferable.

Examples:

- Token is USDT/USDC, address holds funds but is blacklisted then they can still create proposal.
- Token is morpho-vault-v2 share, but holder is blacklisted by gates.

Recommendation: Do not use `token.balanceOf(_who)`, rather actually lock the tokens in `LockManager` by making a transfer.

Aragon: Fixed in PR 28.

Spearbit: Fix verified.

5.2.6 `currentTokenSupply()` can be gamed risk-free to manipulate either side of votes in certain assets

Severity: Medium Risk

Context: MajorityVotingBase.sol#L376-L379

Description: `currentTotalSupply` is being used at following places to check if various thresholds are reached:

- `MajorityVotingBase.isSupportThresholdReachedEarly`:

```
uint256 noVotesWorstCase = currentTokenSupply() - proposal_.tally.yes - proposal_.tally.abstain;

return (RATIO_BASE - proposal_.parameters.supportThresholdRatio) * proposal_.tally.yes
    > proposal_.parameters.supportThresholdRatio * noVotesWorstCase;
```

- `MajorityVotingBase.isMinVotingPowerReached`:

```
uint256 _minVotingPower = _applyRatioCeiled(currentTokenSupply(),
↪  proposal_.parameters.minParticipationRatio);
return proposal_.tally.yes + proposal_.tally.no + proposal_.tally.abstain >= _minVotingPower;
```

- `MajorityVotingBase.isMinVotingPowerReached`:

```
uint256 _minApprovalPower =
    _applyRatioCeiled(currentTokenSupply(), proposals[_proposalId].parameters.minApprovalRatio);
return proposals[_proposalId].tally.yes >= _minApprovalPower;
```

However, in case of following tokens being locked asset:

- ERC4626 shares which can be minted permissionlessly (morpho vault-v2 share, metamorpho share).

- Tokens which can be minted 1:1 (or almost 1:1) for underlying such as various RWA tokens or staked tokens (sUSDC, sUSDT, wstETH, USDe, etc...).
- Tokens with in-built flash mint & burn capabilites (DAI, RAI, FRAX, aTokens, cTokens, various other LP tokens & wrappers, WETH, etc...).

A malicious actor doesn't require external flash loan and neither carry a significant risk to mint & burn such tokens to manipulate the proposal by skewing `totalSupply` when it is read by the contract. It can be bundled atomically in a single block in via PBS (like flashbots) on ethereum. The impact is much higher especially when skewed during the last block of the given deadline of the proposal.

Recommendation:

- Avoid using `currentTotalSupply` where it can be skewed.
- If token supply has increased or decreased by `x%` than last recorded supply, compute using both min and max token supply and take the conservative figures.
- Consider making such cases explicit in the documentation if risk is acceptable.

Aragon: Fixed in PR 47.

Spearbit: Fix verified.

5.3 Low Risk

5.3.1 Avoid public initial-setter function

Severity: Low Risk

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

Summary: The `LockManagerBase` contract has a public `setPluginAddress()` function through which the plugin address is set once during its initialization. This function has no authentication and is at danger of being frontrun by a malicious actor when it is not called within the same transaction that also deployed the contract implementing `LockManagerBase`.

Finding Description: There is a 1-to-1 relationship between the Lock Manager contract and the Plugin contract, both need the address of each other in order to interact with one another. Due to this deployment inter-dependency the `setPluginAddress()` allows first deploying the Lock Manager contract, then the Plugin contract, and finally set the Plugin contract's address within the Lock Manager.

In the normal case this shouldn't be a problem: Once `LockToVotePluginSetup` has deployed the implementation of `LockToVotePlugin`, the `LockToVotePluginSetup.prepareInstallation()` function makes sure that both `setPluginAddress()` and `initialize()` are called on both of the contracts. But this relies on the assumption that `LockToVotePluginSetup` will be used in all cases.

Recommendation: If this function is to be kept, consider adding a warning to it, such as `Must be called within the same transaction as contract creation` to prevent mistakes that invite malicious actors. Preferrably, avoid having a public setter function in the first place by using (as provided by `utils/proxy.sol`) `predictProxyAddress()` to predict the plugin address. `LockManagerERC20` can then be deployed while passing the predicted plugin address into its constructor instead of needing the public setter function. Finally, the `LockToVotePlugin` can be deployed at the predicted address with `createSaltedProxyAndCall()` instead of `createProxyAndCall()` within `LockToVotePluginSetup.prepareInstallation()`.

Aragon: Fixed in PR 36.

Spearbit: Fix verified. Solves the issue by adding authentication to the public setter function, instead of using `CREATE2`.

5.3.2 Avoid Asymmetry in `clearVote()` function

Severity: Low Risk

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

Summary: The `clearVote()` function of the `LockToVotePlugin` contract resets the user's active voting power on the proposal to 0, but it does not reset the vote option back to `None`. Although this does not cause any immediate issue, code asymmetries such as this should be avoided.

Finding Description: The `vote()` function updates voting power and chosen voting option. The option for a new vote specifically is changed from `VoteOption.None` to `VoteOption.Yes`, `VoteOption.No`, or `VoteOption.Abstain`.

```
proposal_.votes[_voter].voteOption = _voteOption;
proposal_.votes[_voter].votingPower = _newVotingPower;
```

But in the `clearVote()` function we find that only the voting power is reset to 0, and the the voting option is left to its previous value. This is despite the function name implying that the vote would be *cleared*.

```
proposal_.votes[_voter].votingPower = 0;
```

Recommendation: Although this has no apparent impact on the user's experience or on the system's security at this moment, we still strongly recommend avoiding asymmetries like these as they may lead to unexpected issues in the future.

```
  proposal_.votes[_voter].votingPower = 0;
+  proposal_.votes[_voter].voteOption = VoteOption.None;
```

Aragon: Fixed in PR 33.

Spearbit: Fix verified.

5.3.3 Early execution via `vote()` bypasses authentication

Severity: Low Risk

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

Summary: Execution of a proposal requires having `EXECUTE_PROPOSAL_PERMISSION_ID` permission. The `vote()` function attempts automatic execution if the vote that was made allows for early execution of the proposal. But we find that it will only be verified that the `LockManager` contract has that permission, instead of the person who actually made the vote.

Finding Description: Normally, proposals that have successfully passed can be executed by calling the public `execute()` function on the plugin contract. The `MajorityVotingBase` contract ensures that `execute()` may only be called by those with the `EXECUTE_PROPOSAL_PERMISSION_ID` permission via the `auth()` function modifier.

Proposals may however be set to allow for early execution. In this case, a proposal may be executed early (before its `endDate`) when enough people have voted Yes or Abstain, such that even if all remaining votes were to be for No, the proposal would still pass at the current moment.

The `vote()` function intends to immediately execute a proposal if the new vote has made the proposal eligible for early execution. While it attempts to verify for the `EXECUTE_PROPOSAL_PERMISSION_ID` permission, it fails to do so properly. The reason is that, while `execute()` is called directly on the plugin contract, the `vote()` function is instead called via the lock contract because it has the voting power balances.

The plugin contract here checks the permissions of the lock contract, instead of the address that originally called the lock contract. It should be noted that, in the vast majority of cases, the `EXECUTE_PROPOSAL_PERMISSION_ID` permission would have been given to `ANY_ADDR`, meaning that any address would have had this permission anyway.

Recommendation: Don't call the `_attemptEarlyExecution()` function in the name of the `msg.sender`, which is enforced to be the lock manager according to the `auth()` modifier, but instead have it check permissions for the passed `_voter` address, that contains the address of the person that called `vote()` on the lock manager contract.

```
  function vote(
      uint256 _proposalId,
      address _voter,
      VoteOption _voteOption,
      uint256 _newVotingPower
```

```
  ) public override auth(LOCK_MANAGER_PERMISSION_ID) {
      Proposal storage proposal_ = proposals[_proposalId];

      // ...

      if (proposal_.parameters.votingMode == VotingMode.EarlyExecution) {
-         _attemptEarlyExecution(_proposalId, _msgSender());
+         _attemptEarlyExecution(_proposalId, _voter);
      }
  }
```

Aragon: Fixed in PR 26.

Spearbit: Aragon has decided to fully remove the `EarlyExecution` voting mode with the PR 26.

5.3.4 Failed early execute attempt in `vote()` can impact proposal outcome

Severity: Low Risk

Context: LockToVotePlugin.sol#L302-L310

Summary: The `vote()` function attempts automatic execution if the vote that was made allows for early execution of the proposal. If the execution of the proposal fails due to external factors, the entire transaction, and therefore also the vote that made the proposal pass, will be reverted. Therefore it's possible that a proposal expires unsuccessfully, despite there having been voting attempts that would have made it pass.

Finding Description: Let's assume the following scenario:

- There is a proposal that is both close to passing and close to ending.
- The proposal's `VotingMode` is `EarlyExecution`.
- Someone attempts making a large vote that would make the proposal pass for both `Standard` and `EarlyExecution` thresholds.
- Executing the proposal right now will fail given the current state of the blockchain, this issue is temporary.

The user attempting to vote will inadvertently trigger the immediate execution of the proposal because of the `_attemptEarlyExecution()` call within the `vote()` function in the plugin contract. An error that happens during execution will bubble up and cause the entire vote to be reverted.

Because the proposal was close to ending, it will now expire unsuccessfully because the attempted vote that would have gotten the proposal above required thresholds was reverted.

Recommendations: One possible solution could be to avoid the "early execution" and simply store the "early-passed" flag of the proposal storage entry. From now on that proposal can be executed by anyone who owns the `EXECUTE_PROPOSAL_PERMISSION_ID` permission. This new flag needs to be considered in functions like (but not limited to) `_canExecute`, `_hasSucceeded` and `_isProposalOpen`, `_canVote` and so on.

Aragon: Fixed in PR 26.

Spearbit: Aragon has decided to fully remove the `EarlyExecution` voting mode with the PR 26.

5.3.5 `LockToVote.vote` and `LockToVote.clearVote` functions should explicitly validate the `msg.sender` to be the `LockManager`

Severity: Low Risk

Context: LockToVotePlugin.sol#L28, LockToVotePlugin.sol#L148, LockToVotePlugin.sol#L211

Description: The `LockToVote.vote` and `LockToVote.clearVote` functions are both calling the `auth(LOCK_MANAGER_PERMISSION_ID)` modifier, and they do not validate that `msg.sender == lockManager`. This means that those functions could theoretically be called by an entity who has the permission but is not the lock manager.

The LockManager contract is the one that tell to the voting plugin both who is the calller and how much voting power (tokens) have been locked. Relative to the vote function, it would allow such an external entity to vote on behalf of any address with an arbitrary amount of voting power without the need to transfer and lock any token (corresponding to the voting power). This gives to that external entity to freely decide the success or failure of any proposals.

Relative to the cleanVote function, it would allow such an external entity to arbitrarily clean the vote of an existing voter without his own consent. Like before this action gives to that external entity to freely decide the success or failure of any proposals.

On the other end, if the LOCK_MANAGER_PERMISSION_ID permission is revoked from the LockManager, it would prevent a working LockManager from casting votes or cleaning an existing one, disrupting the core usage of the voting plugin and the lock manager.

Recommendation: Aragon should add an explicit sanity checks in both the vote and cleanVote functions and revert when msg.sender != address(lockManager).

The LOCK_MANAGER_PERMISSION_ID permission can be left as it is given that it will be removed once the plugin is uninstalled and will correctly prevent the LockManager from casting new votes after the uninstallation process.

Aragon: Fixed in PR 39.

Spearbit: The PR 39 has implemented the following changes:

- LockToVote.clearVote: removed auth(LOCK_MANAGER_PERMISSION_ID) and added the requirement msg.sender == address(lockManager). The LockManager will always be able to clean the vote. Note that, unless there's an upgrade in the code, the lockManager address cannot be changed after deployment.
- LockToVote.vote: added the requirement msg.sender == address(lockManager). The DAO could remove the permission LOCK_MANAGER_PERMISSION_ID from the LockManager and prevent it from casting a vote on behalf of the user.

5.3.6 Prevent or at least document the creation of proposal with empty actions

Severity: Low Risk

Context: LockToVotePlugin.sol#L79

Description: The current implementation of the LockToVotePlugin allows the caller to create proposals that do not contain any actions. If the proposal passes and it is executed the PluginUUPSUPgradeable._execute function will invoke IExecutor.execute(bytes32 _callId, Action[] memory _actions, uint256 _allowFailureMap) in call or delegatecall mode (see PluginUUPSUpgradeable.sol#L171-L216). As far as we can see there's no point to allowing the creation of a proposal that will possibly lock the user's tokens for no real reason.

Recommendation: Aragon should revert the createProposal function when _actions.length == 0. Cantina Managed: The client has acknowledged the issue with the following statement:

Aragon: Some DAOs use governance plugins to get feedback from the community by creating "signaling" proposals. This is a way for protocol teams, curators, etc to do a temperature check before committing to a specific, binding set of actions to submit.

We acknowledge the risk of a user locking tokens to vote on a signaling proposal with LockToVote.

Spearbit: Acknowledged.

5.3.7 Side effects of uninstalling a LockToVotePlugin with active proposals and locked votes

Severity: Low Risk

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

Description: The current uninstallation process of the LockToVotePlugin will not revert if the LockToVotePlugin has ongoing (still open for voting) proposals with already registered votes. Once the plugin has been uninstalled and all the permissions have been revoked, it won't be possible to interact with the LockToVotePlugin anymore.

This behavior will bring some side effects that will affect those users who have already cast votes on proposals that are still "open for vote":

1. These users won't be able to unlock their votes and withdraw their tokens. They must wait for the end of the voting periods of all the proposals they have voted on. At that point they will be able to call `LockManager.unlock` and withdraw their tokens.
2. Users will still be able to call `LockManager.lock` and lock more tokens, effectively transferring them to the `LockManager`. If they have no active proposal they have voted on they will be able to instantly call the `LockManager.unlock` and withdraw their tokens; otherwise, they will need to wait, as explained for the above issue point.

Recommendation: Aragon should implement an "onUninstall" mechanism triggered once the plugin has been uninstalled that would.

1. Prevent the users from locking more tokens via `LockManager.lock`. That function should directly revert when the vote plugin connected to the lock manager has been uninstalled.
2. "clean" the `knownProposalIds` set in the `LockManager` to allow anyone to call `LockManager.unlock` and withdraw their tokens.

Aragon: Assuming that an uninstall proposal passes on the LockToVote plugin:

- We expect no meaningful new proposals to be created on this plugin.
- We expect voters to stop voting in future proposals: they would be able to unlock after the uninstall proposal is settled.

Worst case scenario:

- They decide to vote in a proposal created after the uninstall proposal is created, and before the uninstall is applied.
- These voters will need to wait until `endDate`.

We would prefer to:

- Lower the maximum allowed `proposalDuration` setting to 1 month, maximum.
- Not introducing `onUninstall`, since this requires protocol changes on OSx, and going beyond the scope of this particular plugin.

Spearbit: Aragon has acknowledged the finding and won't implement the recommendations suggested. They have decided to reduce the duration of a proposal from 1 year to a maximum of 31 days with the PR 38. This change is welcome and will alleviate the issue described in the proposal. In the worst-case scenario, users will have their token locked for at most 1 month.

5.3.8 `LockToVotePluginSetup.prepareInstallation` returns permissions that will make the plugin installation process to revert

Severity: Low Risk

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

Description: The current implementation of `LockToVotePluginSetup.prepareInstallation` will include in the list of permissions to be granted during the installation of the plugin the following permission:

```
// createProposalCaller (possibly ANY_ADDR) can create proposals on the plugin
permissions[5] = PermissionLib.MultiTargetPermission({
    operation: PermissionLib.Operation.Grant,
    where: plugin,
    who: installationParams.createProposalCaller,
    condition: minVotingPowerCondition,
    permissionId: impl.CREATE_PROPOSAL_PERMISSION_ID()
});
```

The installation process will revert with the error `GrantWithConditionNotSupported` because the permission is using the operation `Grant` and not `GrantWithCondition` but it has a non-empty condition.

Recommendation: Aragon should update the `prepareInstallation` function with the following change.

```
  // createProposalCaller (possibly ANY_ADDR) can create proposals on the plugin
  permissions[5] = PermissionLib.MultiTargetPermission({
-       operation: PermissionLib.Operation.Grant,
+       operation: PermissionLib.Operation.GrantWithCondition,
      where: plugin,
      who: installationParams.createProposalCaller,
      condition: minVotingPowerCondition,
      permissionId: impl.CREATE_PROPOSAL_PERMISSION_ID()
  });
```

Aragon: Fixed in PR 35.

Spearbit: Fix verified.

5.3.9 Unlocking and removing votes with voting mode `VoteReplacement` is not always coherent

Severity: Low Risk

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

Description: If the user has cast a vote for only one proposal created with the voting mode `VoteReplacement`, he will be able to "remove" that vote and withdraw the tokens by calling `LockManager.unlock` even if the proposal is still open for voting. If instead has also casted a vote on a still open proposal that has not been created via with the voting mode `VoteReplacement` the user will not be able to execute the `LockManager.unlock` to just remove the vote. The operation will revert with the error `VoteRemovalForbidden`. The user won't also be able to "manually" remove the vote by replacing it with a `None` vote.

The above behavior is inconsistent and go against with the one described by the team: a user should always be able to "withdraw" (remove) his vote when the voting mode is `VoteReplacement`.

Recommendation: Aragon should allow the user to "manually" remove and clean his vote if the proposal involved has the voting mode `VoteReplacement`.

> Allowing the user to "remove" the vote could bring side effects in case Aragon will require the user to use the locked token amount to enable the proposal creation. At that point the user could just:

1. Lock tokens.
2. Create the proposal and vote (required to create it).
3. Remove his vote.
4. Unlock the locked tokens.

Aragon: We acknowledge the finding. We consider that "vote replacement" should allow you to replace the sense of your vote (Yes, No, Abstain) with another choice (Yes, No, Abstain), rather than giving the expectation to remove it proactively.

- If you have active standard voting proposals, you cannot unlock. This affects regardless of how many vote replacement proposals you voted for.
- If you only have vote replacement proposals, then you may unlock.
- If you have no votes, you can also unlock.

Spearbit: Acknowledged.

5.3.10 `LockToVotePlugin` should add an upper bound to the proposal actions length

Severity: Low Risk

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

Description: The current implementation of `createProposal` does not perform any sanity check on the length of the action length attached to a proposal. Users could create proposals that will revert during the execution because the loop on the action's execution will revert with an Out Of Gas error.

Recommendation: On the `Dao.sol` contract, for example, the action's length is bound to a max of 256 actions. Aragon should consider using the same upper bound or even better lowering it given that proposals could be created by "normal" token holders and not "authed" users.

Aragon: We acknowledge the finding.

- The gas cost of having 256 or more actions is very high, if not exceeding the gas limit.
- There is no incentive to spend a lot of money to create a proposal that will not execute.
- Even so, the actions can always be submitted as 2+ proposals.
- For veto focused scenarios, we don't expect more than 1~2 actions in total.

Spearbit: Acknowledged.

5.3.11 `isContract` and `_supportsErc20` checks can produce false positives

Severity: Low Risk

Context: LockToVotePluginSetup.sol#L83-L87

Description: The token validation logic using `isContract()` and `_supportsErc20()` can produce false positives, potentially allowing non-compliant or malicious tokens to bypass validation checks. With EIP-7702, EOAs can temporarily have contract code during transaction execution, causing `isContract()` to return `true` for accounts that aren't actually contracts. Also, `isContract` has been deprecated upstream

Additionally, `_supportsErc20()` performs static calls to check for `balanceOf`, `allowance`, and `totalSupply` functions, but malicious contracts can implement these function signatures without proper ERC20 semantics, returning successful responses while implementing completely different logic.

Recommendation:

- Consider improving the natspec and documentation to properly outline the associated risks.
- Maintain a list of verified tokens or framework to assess supported tokens offchain.

Aragon: The goal of this function is to catch basic accidental mistakes, rather than rejecting compliant yet malicious tokens. The choice of the token address is an arbitrary setting that the community either accepts or simply refrains from interacting with it. If a malicious ERC20 token was used for voting, then all of the assumptions on top would become pointless.

We acknowledge the potential risk of the deployer defining a malicious ERC20 token.

Spearbit: Acknowledged.

5.3.12 `currentTokenSupply()` may not represent actual circulating available supply for voting

Severity: Low Risk

Context: LockToVotePlugin.sol#L249

Description: `currentTokenSupply()` varies largely in different contexts, for e.g:

- If tokens are minted but are in vesting or staking contract, they can't be used but are counted.
- Same as above, if tokens are already owned by dao / treasury they can't contribute to voting.

- If two standard proposals are live in parallel, tokens used for one can't be used for others.

This implies "Minimum Participation" formulae from specs is not correct representation and interpretation of supply available to be used for voting. This can mislead users by wrongly gauging vote weight of the user in participation. Various thresholds in vote settings does reduce the risk to some extent but don't fully mitigate the core problem of unknown part of supply which can't be used in voting.

Recommendation:

- Do not rely on `currentTokenSupply()`, and if it possible to figure out "usuable supply" that should be used instead.
- Consider making such cases explicit in the documentation.

Aragon: Fixed in PR 34.

Spearbit: Fix verified.

5.4 Gas Optimization

5.4.1 Various gas efficiency improvements

Severity: Gas Optimization

Context: MajorityVotingBase.sol#L515-L545, LockToVotePlugin.sol#L107-L112

This issue is a collection of places where we have found possible gas efficiency improvements.

- MajorityVotingBase.sol#L515-L545: The functions `isSupportThresholdReachedEarly`, `isMinVotingPowerReached` and `isMinApprovalReached` are all fetching the `currentTokenSupply()` which makes 2 external calls (one to fetch the token address from the lock manager and one to fetch the token's total supply).

  These functions can be refactored to accept the `tokenTotalSupply` as an arbitrary input. At this point, it can probably also take `proposal_` as `memory` instead of `storage`, given that you have already loaded it before.

- LockManagerBase.sol#L242-L244:

```
unchecked {
  _i++;
}
```

  Since Solidity 0.8.22 there should be no need for `unchecked { i++; }` optimizations in `for` loops anymore. The version used in this project according to the foundry.toml file is `solc = "0.8.28"`.

  The same issue exists in LockToVotePlugin.sol#L149-L111.

- LockToVotePlugin.sol#L107-L112:

```
proposal_.parameters.votingMode = votingMode();
proposal_.parameters.supportThresholdRatio = supportThresholdRatio();
proposal_.parameters.startDate = _startDate;
proposal_.parameters.endDate = _endDate;
proposal_.parameters.minParticipationRatio = minParticipationRatio();
proposal_.parameters.minApprovalRatio = minApprovalRatio();
```

  Each of these function alls accesses the global voting settings in order to store a copy of them within the proposal. Consider loading the settings in a local `memory` variable instead of always loading them from the contract's storage to save gas.

- LockToVotePlugin.sol#L77-L83:

```
function createProposal(
  bytes calldata _metadata,
  Action[] memory _actions,
  uint64 _startDate,
  uint64 _endDate,
```

```
    bytes memory _data
  ) external auth(CREATE_PROPOSAL_PERMISSION_ID) returns (uint256 proposalId) {
```

Consider changing the datalocation of `_actions` and `_data` from `memory` to `calldata` to avoid a likely unnecessary copy-to-memory step when the function is called.

Aragon:

- `currentTokenSupply()` grouping: Acknowledged. We'd like to keep functions as they are for interoperability within other scenarios.
- Unchecked iterator: See PR 51.
- Proposal parameters: Acknowledged.
- Create proposal: Acknowledged. We want to remain compatible with the IProposal interface as it is.

Spearbit: Fix verified.

5.5 Informational

5.5.1 Ensure specific Solidity Version

Severity: Informational

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

Summary: We find various Solidity version pragmas within the source files:

- `pragma solidity ^0.8.8;`.
- `pragma solidity ^0.8.8;`.
- `pragma solidity ^0.8.17;`.

Whereas the actually used version according to the local `foundry.toml` is is `0.8.28`.

Recommendation: It's generally considered best practice to avoid floating pragmas (`^`) or wide version ranges for contracts that will be deployed, and instead use the pragma to lock the compilation, tests, and deployment to a single definite Solidity version (`pragma solidity 0.8.28;`).

Wider version ranges should be restricted to contracts, libraries, and interfaces that may be imported and used in other projects that are likely to use a different Solidity compiler version.

Aragon: Fixed in PR 45.

Spearbit: Fix verified.

5.5.2 Use `require()` instead of `if`-statements

Severity: Informational

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

Summary: The codebase makes use of statements in the form of `if (condition) revert CustomError()`. The use of `require()` statements improves the readability of such statements while, since Solidity 0.8.26, retaining the ability to use custom errors.

Recommendation: Adopt `require()` statements in place of if-statements throughout the code base.

```
- if (condition) revert CustomError();
+ require(!condition, CustomError());
```

Though note that this switch should be executed with extreme caution as it requires the negation of the condition's expression, eg. `a || (b == c && d > 0)` to `!a && (b != c || d <= 0)` which is sometimes difficult and error-prone.

Aragon: Though it is a good practice, we prefer leave it as it is now.

Spearbit: Acknowledged.

5.5.3 Prefer upscaling over downscaling

Severity: Informational

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

Summary: Currently, during threshold checks for participation and approval, downscaling with division is used. This can be avoided by instead upscaling the other side of the inequality.

Finding Description: The following is a LaTeX representation of the threshold checks as they are implemented in the code. For both the participation and the approval checks, the threshold ratio (which is upscaled) is multiplied with the total amount of voting power in existence (N_{total}). The result of this calculation is then downscaled by dividing through `RATIO_BASE`.

This can be avoided, similarly to how it is implicitly avoided in the support threshold check: Both sides are upscaled when they are compared, therefore, neither side needs to be downscaled.

Support Threshold Math:

$$\left(\overbrace{1,000,000}^{\text{RATIO\_BASE}} - \text{supportThresholdRatio}\right) \cdot N_{\text{yes}} > \text{supportThresholdRatio} \cdot N_{\text{no}}$$

Participation Threshold Math:

$$N_{\text{yes}} + N_{\text{no}} + N_{\text{abstain}} \geq \frac{N_{\text{total}} \cdot \text{minParticipationRatio}}{\underbrace{1,000,000}_{\text{RATIO\_BASE}}}$$

Approval Threshold Math:

$$N_{\text{yes}} \geq \frac{N_{\text{total}} \cdot \text{minApprovalRatio}}{\underbrace{1,000,000}_{\text{RATIO\_BASE}}}$$

Early Execution Support Threshold Math:

$$\left(\overbrace{1,000,000}^{\text{RATIO\_BASE}} - \text{supportThresholdRatio}\right) \cdot N_{\text{yes}} > \text{supportThresholdRatio} \cdot \left(N_{\text{total}} - N_{\text{yes}} - N_{\text{abstain}}\right)$$

Recommendation: Avoid downscaling in both of the following functions by simply upscaling the other side appropriately:

```
  /// @inheritdoc IMajorityVoting
  function isMinVotingPowerReached(uint256 _proposalId) public view virtual returns (bool) {
      Proposal storage proposal_ = proposals[_proposalId];

-     uint256 _minVotingPower = _applyRatioCeiled(currentTokenSupply(),
↪  proposal_.parameters.minParticipationRatio);
+     uint256 _minVotingPower = currentTokenSupply() * proposal_.parameters.minParticipationRatio;

      // The code below implements the formula of the
      // participation criterion explained in the top of this file.
      // `N_yes + N_no + N_abstain >= minVotingPower = minParticipationRatio * N_total`
```

```
-     return proposal_.tally.yes + proposal_.tally.no + proposal_.tally.abstain >= _minVotingPower;
+     return RATIO_BASE * (proposal_.tally.yes + proposal_.tally.no + proposal_.tally.abstain) >=
↪  _minVotingPower;
  }

  /// @inheritdoc IMajorityVoting
  function isMinApprovalReached(uint256 _proposalId) public view virtual returns (bool) {
-      uint256 _minApprovalPower = _applyRatioCeiled(currentTokenSupply(),
↪  proposals[_proposalId].parameters.minApprovalRatio);
+     uint256 _minApprovalPower = currentTokenSupply() *
↪  proposals[_proposalId].parameters.minApprovalRatio;

-     return proposals[_proposalId].tally.yes >= _minApprovalPower;
+     return RATIO_BASE * proposals[_proposalId].tally.yes >= _minApprovalPower;
  }
```

Aragon: Fixed in PR 46.

Spearbit: Fix verified.

5.5.4 Various missing/incorrect comments

Severity: Informational

Context: LockManagerBase.sol#L38, MajorityVotingBase.sol#L152-L153, MajorityVotingBase.sol#L195-L196, ILockManager.sol#L23, ILockManager.sol#L25, LockToVotePlugin.sol#L94

Description: This issue is a collection of places where we have found natspec/inline comments to be lacking.

- MajorityVotingBase.sol#L195-L196:

```
/// @param startDate The start date of the proposal vote.
/// @param endDate The end date of the proposal vote.
```

  It would be useful to clarify the interval within which the user can vote, it being `[startDate, endDate)` (ie. the `endDate` is not included in the range, allowing users only to vote until `endDate - 1`).

- LockToVotePlugin.sol#L94:

```
/// @dev `minProposerVotingPower` is checked at the the permission condition behind
↪  auth(CREATE_PROPOSAL_PERMISSION_ID)
```

  This comment could be relocated above the `function createProposal(`, below the `/// @dev Requires the` CREATE_PROPOSAL_PERMISSION_ID `permission` natspec documentation. Its current location is confusing, making it seemingly related to the function below (`_validateProposalDates`).

- ILockManager.sol#L25-L26:

```
/// @notice Returns the current settings of the LockManager.
function settings() external view returns (PluginMode pluginMode);
```

  Missing natspec documentation for `@return`.

- MajorityVotingBase.sol#L30-L35:

```
/// We define two parameters
/// $\texttt{support} = \frac{N_\text{yes}}{N_\text{yes} + N_\text{no}} \in [0,1]$
/// and
/// $\texttt{participation} = \frac{N_\text{yes} + N_\text{no} +
↪  N_\text{abstain}}{N_\text{total}} \in [0,1],$
/// where $N_\text{yes}$, $N_\text{no}$, and $N_\text{abstain}$ are the yes, no, and abstain
↪  votes that have been
/// cast and $N_\text{total}$ is the total voting power available at proposal creation time.
```

This is lacking a description of the third parameter, which is `approval`.

$$\texttt{approval} = N$$

$\text{yes}_{\overline{N}}$

total $\in$[0,1]

ILockManager.sol#L23:

```
/// @notice Helper contract acting as the vault for locked tokens used to vote on multiple plugins and
↪  proposals.
```

The comment is misleading, as you *cannot* vote on multiple plugins but *only* on the one configured (which can't change). It can make sense to simplify it given that right now it only supports "Vote Plugin".

MajorityVotingBase.sol#L152-L153:

```
/// @param supportThresholdRatio The support threshold ratio.
///     Its value has to be in the interval [0, 10^6] defined by `RATIO_BASE = 10**6`.
```

The comment here is incorrect; the `supportThresholdRatio` value is in the interval of `[0, 10^6-1]` or you could express it as `[0, 10^6)`.

LockManagerBase.sol#L153:

```
- @dev Both plugins already enforce unicity
+ @dev Plugin already enforces unicity
```

Comment is outdated, as there is now only a single plugin.

ILockManager.sol#L9-L10:

```
/// @notice Defines wether the voting plugin expects approvals or votes
///     Placeholder for future plugin variants
```

This dev comment is incorrect; the existing code does *only* support *vote* types of plugins (no *approval*). The `LockToApprovePlugin` contract has been removed.

ILockManager.sol#L90-L92:

```
/// @notice Called by the lock to vote plugin whenever a proposal is executed (or ended). It instructs
↪  the manager to remove the proposal from the list of active proposal locks.
/// @param proposalId The ID of the proposal that msg.sender is reporting as done.
function proposalEnded(uint256 proposalId) external;
```

This should mention that there's no guarantee that `proposalEnded()` will be reliably called for a proposal ID and that manually checking a proposal's state may be necessary to notice that it has ended.

ILockToGovernBase.sol#L32-L35:

```
/// @notice Returns wether a proposal is open or not.
/// @param _proposalId The ID of the proposal.
/// @return True if the proposal is open, false otherwise.
function isProposalOpen(uint256 _proposalId) external view returns (bool);
```

This function has been accidentally used as if it is `isProposalStillOpen()` in some places. Consider adding a comment clarifying that this returns false for proposals that have not opened *yet*.

ILockToVote.sol#L12-L17:

```
/// @notice Checks if an account can participate on a proposal. This can be because the vote
/// - has not started,
/// - has ended,
/// - was executed, or
/// - the voter doesn't have voting powers.
/// - the voter can increase the amount of tokens assigned
```

"This can be" seems incorrect here. "This can fail" seems more accurate.

ILockToVote.sol#L38-L41:

```
/// @notice Reverts the existing voter's vote, if existing.
/// @param proposalId The ID of the proposal.
/// @param voter The voter's address.
function clearVote(uint256 proposalId, address voter) external;
```

Should mention that the function reverts with an error if a vote cannot be cleared due to voting settings. This is an important aspect to the token locking functioning correctly.

LockManagerBase.sol#L36-L38: Add a `@dev` comment that underlines the fact that the `ProposalEnded` event could be triggered with a "delay" compared to the "real" proposal end period.

Aragon: Fixed in PR 58.

Spearbit: Fix verified.

5.5.5 Various missing/lacking sanity checks

Severity: Informational

Context: LockManagerBase.sol#L170, LockToGovernBase.sol#L61, MajorityVotingBase.sol#L305, ILockManager.sol#L9-L10, LockManagerERC20.sol#L19, LockToVotePlugin.sol#L121-L125, LockToVotePlugin.sol#L150

Description: This issue is a collection of places where we have found sanity checks to be lacking.

- LockToVotePlugin.sol#L121-L126:

  This should revert when `_actions[i].to == address(0)`.

- MajorityVotingBase.sol#L647-L650:

  Consider adding checks to prevent nonsensically long proposal durations. Note that the `votingSettings.proposalDuration` variable's naming is rather misleading, as its name seems to indicate how long a proposal may be configured to run. But instead it is actually the minimum that a proposal must last for, which can be configured anywhere from 60 minutes to 1 year. If possible, consider renaming the setting to `minProposalDuration` for accuracy.

- LockManagerBase.sol#L170:

  When setting the plugin address within the lock manager, consider reverting if `_newPluginAddress.lockManager() != address(this)`.

  Further, consider adding the `DAO` contract to the state variable of `LockManagerBase` and enforce that `plugin.dao() == dao` (from the lock manager).

  One could also verify that the vote plugin has not been "already" used somewhere else, but on the vote plugin the proposals are stored in `mapping(uint256 => Proposal) internal proposals` which can't be queried to get the length value. If this appears valuable to check, a `lastProposalId` or `proposalCount` could be added to the plugin and fetched to be validated.

- LockManagerERC20.sol#L18-L20:

  Although this check is already performed in the setup process, consider reverting if `erc20Token == address(0)` or if `erc20Token` is *not* an already deployed contract (contract size == 0).

- LockToVotePlugin.sol#L150-L154:

  If the `proposal_` does not exist the `_canVote` will return `false` and so it will revert with the `VoteCastForbidden` error. While from a security prospective this is correct and fine, the more appropriate behavior in this case would be to perform the same check already performed in `canVote` and be consistent with that.

  ```
  if (!_proposalExists(_proposalId)) {
      revert NonexistentProposal(_proposalId);
  }
  ```

  Consider performing this in every function that needs to interact with a proposal if it's not already done.

- LockToGovernBase.sol#L61:

  Although already performed during plugin setup, consider adding a sanity check validating if the `lockManager` has been configured with the `PluginMode.Voting`: `require(lockManager.settings() == PluginMode.Voting)`.

- MajorityVotingBase.sol#L305:

  Consider adding a sanity check reverting when the `_dao` passed into the constructor is `address(0)`. Neither `__PluginUUPSUpgradeable_init` nor `__DaoAuthorizableUpgradeable_init` currently revert if `_dao` is the empty address.

- LockManagerBase.sol:

  Although ensured by the plugin, consider verifying that a given `_proposalId` hasn't already been added to the `knownProposalIds`:

  ```
  bool added = knownProposalIds.add(_proposalId);
  require(added, new ProposalAlreadyAddedError());
  ```

Aragon: Applied some of the recommendations in PR 38 and PR 52.

Spearbit: Recap of the changes:

- Proposal max duration has been upper bounded to max 31 days via PR 38.
- Proposals with actions with an empty target (`_actions[i].to`) will be rejected via PR 52.

All the remaining recommendations have been acknowledged by the project and won't be fixed.

5.5.6 Various Unused/Unreachable/Dead Code issues

Severity: Informational

Context: MajorityVotingBase.sol#L560-L567

Description: This issue is a collection of places where we have found code to be unreachable or unused and recommend removal.

- ILockManager.sol#L7:

  ```
  import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
  ```

  This import is unused and can be removed.

- ILockToGovernBase.sol#L5-L9:

  ```
  import {IDAO} from "@aragon/osx-commons-contracts/src/dao/IDAO.sol";
  import {Action} from "@aragon/osx-commons-contracts/src/executors/IExecutor.sol";
  import {PluginUUPSUpgradeable} from
  ↪  "@aragon/osx-commons-contracts/src/plugin/PluginUUPSUpgradeable.sol";
  import {IPlugin} from "@aragon/osx-commons-contracts/src/plugin/IPlugin.sol";
  ```

  These imports are unused and can be removed.

- LockToVotePluginSetup.sol#L33-L65:

```
LockManagerERC20 private immutable lockManagerImpl;
```

This private, immutable variable is initialized within the constructor but then never used again. Note also that right after the creation of a `LockManagerERC20` the public `setPluginAddress()` should be called, which the constructor currently doesn't do. This seems to be a leftover and should be removed.

- MajorityVotingBase.sol#L560-L567:

This check is supposed to handle the edge case when the `_execute()` function is called for a proposal that has not started yet. But the check is unreachable since, to be reached, the threshold checks would need to be passed. Which isn't possible even with all thresholds set to 0 because one would still need 1 wei for the support threshold to pass.

Consider removing this unreachable check or moving it up to the top of the function.

- LockToVotePlugin.sol#L12:

```
import {ERC165Upgradeable} from
↪  "@openzeppelin/contracts-upgradeable/utils/introspection/ERC165Upgradeable.sol";
```

This import is unused and can be removed.

- LockToVotePlugin.sol#L18:

```
using SafeCastUpgradeable for uint256;
```

No safe casting operation appears to be used within this contract.

- LockToGovern.sol#L8-L9:

```
import {ILockManager} from "../interfaces/ILockManager.sol";
import {ILockManager} from "../interfaces/ILockManager.sol";
```

`ILockManager` is imported twice.

- LockToVotePluginSetup.sol#L5-L30:

```
import {Clones} from "@openzeppelin/contracts/proxy/Clones.sol";
import {ERC165Checker} from "@openzeppelin/contracts/utils/introspection/ERC165Checker.sol";
// ...
    using Clones for address;
    using ERC165Checker for address;
```

Neither of these libraries appear to be used.

Aragon: Fixed in PR 50.

Spearbit: Fix verified.

5.5.7 After approval, the execution of a proposal has no further validation or restriction

Severity: Informational

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

Description: Once a proposal has "passed", it can be executed without any further validation or restriction:

1. There is no "max retry" upper bound limit if the execution fails.
2. There is no "ordering" concept: `proposal_1` could be executed after `proposal_2` even if `proposal_1` will override the changes made by `proposal_2`.
3. There's no "deadline" to the execution of a proposal.

Recommendation: Aragon should consider implementing the above features or at least explicitly disclosing the missed implementation of them.

Aragon: Following a "chain of proposals" model could allow a whale to submit malicious proposals which blocked the execution of legitimate ones until it was too late for the DAO to act.

We consider this to be outside of the scope, so we acknowledge the feedback.

Spearbit: Acknowledged.

5.5.8 Consider restricting the usage and configuration of the `LockToVotePlugin`

Severity: Informational

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

Description: The `LockToVotePlugin` has been implemented and can be configured (via `LockToVotePluginSetup`) as a general-purpose voting plugin that follows the same "general-purpose" approach as the overall `OSx` framework.

For example the plugin could be deployed and configured to execute proposal toward a contract that is not the `DAO.sol` contract and execute them via `delegatecall` instead of a common low-level `call`.

Recommendation: Given the specific scope of `LockToVotePlugin` and `LockManager`, Aragon should consider to enforce at `LockToVotePluginSetup` or even better directly in the `LockToVotePlugin` to be configured as it "should be intended".

- `_targetConfig.target` should be equal to the `dao` address.
- `_targetConfig.operation` should be equal to `Operation.call`.

Aragon: We acknowledge this finding. We prefer giving the community the ability to use this plugin in other scenarios like within the SPP (plugin orchestrator), which is not a DAO.

Spearbit: Acknowledged.

5.5.9 `createProposals` allows the creation of "identical proposal" in the same block

Severity: Informational

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

Description: The current implementation of `createProposal` reverts if the caller generates a proposal with an ID that would clash (and override) an existing one:

```
proposalId = _createProposalId(keccak256(abi.encode(_actions, _metadata)));
if (_proposalExists(proposalId)) {
    revert ProposalAlreadyExists(proposalId);
}

/// @notice Creates a proposal Id.
/// @dev Uses block number and chain id to ensure more probability of uniqueness.
/// @param _salt The extra salt to help with uniqueness.
/// @return The id of the proposal.
function _createProposalId(bytes32 _salt) internal view virtual returns (uint256) {
    return uint256(keccak256(abi.encode(block.chainid, block.number, address(this), _salt)));
}
```

The uniqueness of a proposal and it's relative ID is given mainly by `block.number` and by the `_salt` that is derived by the values (input parameters createProposals) of of `_actions` and `_metadata`. With the above logic, we know that the creator won't be able to create a proposal that will override an existing one, but it does not prevent the proposer from creating proposals that could be basically identical given that we could:

- Provide the same actions but change the orders.

- Provide the very same actions (with the same order) but provide different metadata, which is ininfluent relative to the proposal execution.

Recommendation: Aragon should consider reverting the `createProposal` if the execution of the input actions would produce the same result of an proposal that has already been saved for the very same block. If this behavior is is instead allowed and intended, it should be documented.

Aragon: We acknowledge this finding:

- The goal behind salted proposal ID's is to avoid replay attacks in cross chain scenarios (i.e. using the approval for proposal `1` in a different chain ID).
- Submitting a proposal with the same actions shuffled could technically produce a different outcome.
- There is no expectation given regarding the unicity of proposals. The creator will need 2x locked tokens for the second proposal and the community will approve or reject, regardless of the proposal similarity.

Spearbit: Acknowledged.

5.5.10 Consider moving structs, events, and error declarations from the contracts to the corresponding interfaces

Severity: Informational

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

Description: The vast majority of the structs, events, and errors are currently defined in the contracts files. While this is not a security issue per se, it makes the code of the contract itself longer and "crowded". By moving those definitions directly to the corresponding interface, the readability and complexity (of already complex contracts) will be drastically improved.

Recommendation: Aragon should consider refactoring the whole code base by moving structs, events, and error declarations from the contracts to the corresponding interfaces.

Aragon: Acknowledged. We will leave this change for a future version.

Spearbit: Acknowledged.

5.5.11 Consider removing any reference and checks relative to `PluginMode.Voting` from the `LockManager`

Severity: Informational

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

Description: While the `LockManager` could in the future support multiple plugins (with different voting logic), the current implementation does only support `PluginMode.Voting`. Given such context and the fact that the `LockManager` is a non-upgradable contract, the codebase could be cleaned and assume that the only possible voting plugin to be deployed and configured with will indeed be a plugin of type `PluginMode.Voting`.

Recommendation: Aragon could:

1. Remove any check to `settings.pluginMode != PluginMode.Voting` across the contract (this is present in multiple places).
2. Enforce on the `setPluginAddress` function that `_newPluginAddress` is indeed of type `PluginMode.Voting` (only allow plugins that support the `ILockToVote` interface).
3. Enforce in the `constructor` of the `LockManager` `settings.pluginMode = PluginMode.Voting;`, and remove the input parameter.

The change should be made in the following functions:

- `lockAndVote(uint256 _proposalId, IMajorityVoting.VoteOption _voteOption)`.
- `lockAndVote(uint256 _proposalId, IMajorityVoting.VoteOption _voteOption, uint256 _amount)`.
- `vote(uint256 _proposalId, IMajorityVoting.VoteOption _voteOption)`.

Note that once the state variable `plugin` has been configured, it cannot be updated anymore.

Aragon: Fixed in PR 49.

Spearbit: Fix verified.

5.5.12 Consider renaming the `ProposalEnded` event to `ProposalExecuted`

Severity: Informational

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

Description: The `LockManager.proposalEnded` function is invoked by the `LockToVotePlugin` contract once a proposal has been executed and not when the "voting period" has ended. Usually it's true that to be able to execute a proposal the vote period must have ended and the voting result must be a success, but there's also the `EarlyExecution` voting mode that allow the proposal to be executed before that the period has ended.

Recommendation: Aragon should consider performing the following changes:

- Rename the `ProposalEnded` event to `ProposalExecuted`.
- Rename the `proposalEnded` function to `proposalExecuted`.

Aragon: Fixed in PR 37.

Spearbit: Fix verified.

5.5.13 Consider improving the `lock` behavior documentation

Severity: Informational

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

Description: Let's assume that the user has locked `1e18` tokens and voted on a proposal that is in `Standard` voting mode, or at least that can't be "unlocked" anytime soon. If the user locks (without voting) more tokens, those tokens will be stuck anyway until the "active" proposals end. There's no way for the user to unlock the part of the tokens that have been locked in the `LockManager` contract but have not been "used" into the voting system. Let's make it even more clear.

1. Alice lock `10e18` USDC.
2. Alice votes `Yes` on the `proposal_1`. Alice cannot unlock and withdraw those `10e18 USDC` until the proposal ends.
3. Alice calls `lock(100e18 USDC)`.

Now those `100e18` USDC are also locked and cannot be withdrawn even if they are not used in any of the proposals as "active" voting power.

Recommendation: Aragon should consider documenting this behavior or require that the user can perform `lockAndVote` functions.

Aragon: Acknowledged.

Spearbit: Aragon has decided to acknowledge the issue and improve the documentation relative to this behavior with the PR 44.

5.5.14 Proposals cannot be "vetoed" after creation

Severity: Informational

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

Description: The current logic of the `LockToVotePlugin` does not consider any "guardian" or "sentinel" role that would be allowed to "early terminate" (veto) a proposal created by a malicious actor. Proposals that have been

created in the voting mode `EarlyExecution` could be "early executed" without providing enough time for the "NO" part of the token holder to express their vote and block the approval and early execution of such proposal.

Recommendation: If this is the intended behavior, Aragon should consider explicitly documenting such behavior.

Aragon:

Regarding the guardian/sentinel:

- We acknowledge this situation.
- The DAO with LockToVote itself are meant to act as the "guardian" of external protocol changes.

Regarding Early Execution:

- This voting mode has been removed, so the attack surface loses most of its span.

Spearbit: Acknowledged.

5.5.15 Creation of a Token Integration Checklist

Severity: Informational

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

The `LockManagerERC20` contract of the Lock To Vote plugin can be configured to hold arbitrary ERC20 tokens. However, not all of the different types of tokens that exist are actually fully supported without causing issues to the system. We recommend creating a Token Integration Checklist and adding it to the plugin documentation to enable future users of the plugin to make an educated decision on whether the token they are planning to use for their governance is actually compatible with the plugin.

Aragon may use the following information as an inspiration for their actual checklist once development of the plugin has been finalized.

- Only ERC20s: This plugin currently only deals with underlying tokens of ERC-20 compatible standards. There exist other fungible token standards such as ERC-1155, but these are not supported.
- Supports Double-Entry-Point Tokens, ie. tokens that share the same tracking of balances but have two separate contract addresses from which this balances can be controlled. Typically protocols that have sweeping functions (for rescuing funds) are vulnerable to these since they bypass checks preventing sweeping of underlying funds. The plugin does not appear to be vulnerable to this and using such tokens as underlying should not cause any issues.
- Fixes required for Non-Reverting Tokens: ERC-20 Tokens historically handle errors in two possible ways, they either revert on errors or they simply return a false boolean as a result. At the time of audit, the plugin did not make use of SafeERC20 and therefore did not support non-reverting tokens without introducing severe security issues.
- Supports ERC20s lacking `decimals()`: Within the ERC-20 standard, the existence of a `decimals()` function is optional. The plugin has no need for this function's existence and supports tokens without it.
- Supports Tokens with Callbacks: There exist various standard extensions such as ERC-223, ERC-677, ERC-777, etc., as well as custom ERC-20 compatible token implementations that call the sender, receiver, or both, during a token transfer. Furthermore, such implementations may choose to call before or after the token balances were updated. This is especially dangerous since it may allow re-entering the protocol and exploit incomplete state updates. The plugin has no apparent issues with such tokens.
- NO support of Deflationary/Inflationary or Rebasing Tokens: There are tokens (such as Aave's aToken) which increase in balance over time, or decrease in balance over time (various algorithmic stable coins), this may cause accounting issues within smart contracts holding them. When a user adds funds to the plugin, they are registered in `lockedBalances[msg.sender]`. The plugin assumes that the sum of balances tracked within stays equal to the actual balance the plugin holds. If the balance increases, the surplus is inaccessibly lost. If the balance decreases, those who withdraw (call `unlock()`) first will obtain their full balance, but not all users will be able to withdraw once funds have run out.

- NO support of Tokens with Transfer Fees: There are tokens which may charge a fee for transfers. This fee could be applied on the value being sent, decreasing the amount reaching the receiver, or it could be applied on the sender's remaining balance. The plugin assumes that the value specified as `amount` during the transfer is exactly that value that actually arrived at the plugin, unless the transfer reverted. It does currently not correct for when the actual received balance deviates.
- Supports Tokens with strict Allowance Handling: There exist tokens which error when attempting to change an existing token allowance from one non-zero value to another non-zero value. The plugin makes no calls to the token's `approve()` function and should therefore not be affected.
- Supports Non-Standard Decimals: Tokens typically have 18 decimals, but some deviate from this, usually towards lower numbers. The plugin supports tokens that have large deviations from the typical 18 decimals. It is only for extremely large decimal numbers (>50), combined with large transfer amounts, that there may be problems due to the 10^6 scaling used.
- Care required for Tokens with variable Supply: The plugin relies on the Total Token Supply to be somewhat stable in order to determine the total existing voting power and use it for threshold checks. Tokens that have burn and mint functionality, or other ways allowing to easily manipulate the total supply, should be integrated with care.

Consider the consulting the Crytic Checklist (Trail of Bits) for further inspiration.

Aragon: Fixed in:

- PR 41.
- PR 53.
- PR 61.
- PR 63.

Spearbit: The recommendations have been implemented in the aforementioned PRs.

5.5.16 Bulk missing natspec documentation

Severity: Informational

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

Description:

- ILockManager.sol#L25-L26: The `settings` function misses the `@return` natspec documentation.
- ILockManager.sol#L36-L37: The `getLockedBalance` function misses the natspec documentation for the `account` parameter and the return value.
- ILockManager.sol#L57-L59: The `vote` function misses the natspec documentation for the `vote` parameter.
- ILockManager.sol#L87-L88: The `setPluginAddress` function misses the natspec documentation for the `_plugin` parameter.
- ILockToGovernBase.sol#L15-L16: The `lockManager` function misses the natspec documentation for the return value.
- LockManagerBase.sol#L30-L31: The `BalanceLocked` event misses the natspec documentation for the `voter` and `amount` parameters.
- LockManagerBase.sol#L33-L34: The `BalanceUnlocked` event misses the natspec documentation for the `voter` and `amount` parameters.
- LockManagerBase.sol#L63-L64: The `knownProposalIdAt` function misses the natspec documentation for the `_index` parameter and the return value.
- LockManagerBase.sol#L68-L69: The `knownProposalIdsLength` function misses the natspec documentation for the return value.

- LockToGovernBase.sol#L17: The `LockManagerDefined` event misses the natspec for the `lockManager` parameter.
- LockToGovernBase.sol#L19: The `NoVotingPower` error misses the natspec documentation.
- LockToGovernBase.sol#L20: The `LockManagerAlreadyDefined` error misses the natspec documentation.
- MajorityVotingBase.sol#L440-L441: The `getVotingSettings` function misses the natspec documentation for the return value.
- LockToVotePlugin.sol#L30: The `VoteCleared` event misses the natspec documentation for the `proposalId` and `voter` parameters.
- LockToVotePlugin.sol#L32: The `VoteRemovalForbidden` error misses the natspec documentation for the `proposalId` and `voter` parameters.

> Some external/public functions in the existing contracts have "local" natspec documentation: move the natspec to the corresponding `interface` file and replace the existing one with the correct `@inheritdoc` tag. Not all the internal/private functions are covered with natspec documentation. This part is not required but more than welcome. It helps both the developer and security researcher to better understand and read the logic of the function.

Recommendation: Aragon should consider addressing all the recommendations made in the above section.

Aragon: Fixed in PR 59.

Spearbit: Fix verified.

5.5.17 Suggestions on Events and missed `indexed` declarations

Severity: Informational

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

Description: The following event could declare their input parameters as `indexed`:

- LockManagerBase.sol#L31: The `BalanceLocked` event could declare the `voter` parameter as `indexed`.
- LockManagerBase.sol#L34: The `BalanceUnlocked` event could declare the `voter` parameter as `indexed`.
- LockManagerBase.sol#L38: The `ProposalEnded` event could declare the `proposalId` parameter as `indexed`.
- LockToGovernBase.sol#L17: The `LockManagerDefined` event could declare the `lockManager` parameter as `indexed`.
- LockToVotePlugin.sol#L30: The `VoteCleared` event could declare the `proposalId` and `voter` parameters as `indexed`.

Recommendation: Aragon should consider declaring the above parameter as `indexed` if they think that they could be a useful data point information for their external dApps/services/monitoring tools.

Aragon: Fixed in PR 48.

Spearbit: Fix verified.