BENQI Governance Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Giovanni Di Siena

ChainDefenders (0x539 & PeterSR)

Jorge

November 10, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The BENQI governance contracts integrates vote-escrow gauges and optimistic token voting to coordinate incentive emissions for both core and ecosystem markets. Modular governance is enabled through Aragon OSx infrastructure which ensures flexible, transparent, and on-chain coordination between liquidity incentives and protocol policy management.

5 Audit Scope

The audit scope was limited to:

```
benqi-governance/src/contracts/BenqiCoreModule.sol
benqi-governance/src/contracts/BenqiEcosystemModule.sol
benqi-governance/src/contracts/CallerId.sol
benqi-governance/src/contracts/Clock.sol
benqi-governance/src/contracts/DistributionManager.sol
benqi-governance/src/contracts/DistributionManagerSetup.sol
benqi-governance/src/contracts/ExecuteSelectorCondition.sol
benqi-governance/src/contracts/GaugeRegistrar.sol
benqi-governance/src/contracts/SpeedCalculator.sol
benqi-governance/src/contracts/UnifiedBudgetAllocator.sol
```

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the BENQI Governance smart contracts provided by BENQI. In this period, a total of 21 issues were found.

This review yielded 11 low severity findings, 5 informational findings, and 5 gas optimizations. The low severity findings were related to edge cases in epoch timing, reward distribution, gauge registration, role authentication, and interface compliance.

Summary

Project Name	BENQI Governance
Repository	benqi-governance
Commit	ded42b671f11...
Audit Timeline	Sept 29th - Oct 10th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	11
Informational	5
Gas Optimizations	5
Total Issues	21

Summary of Findings

[L-01] Distribution logic can be broken by upgrades to the `GaugeRegistrar` by the DAO to add logic to modify the `enableUpdateVotingPowerHook` configuration	Resolved
[L-02] Missing interface compliance validation in `DistributorManager`	Acknowledged
[L-03] `BenqiEcosystemModule` and `UnifiedBudgetAllocator` are not fully ERC-165 compliant	Acknowledged
[L-04] Maximum number of Aragon DAO actions is not checked within the `DistributorManager`	Acknowledged
[L-05] Budget is always calculated using the full `epochDuration` regardless of the actual timing of distributions	Acknowledged
[L-06] Single `rewardToken` design is incompatible with multi-reward emission support	Acknowledged
[L-07] Reward distribution can be skipped when executed within the vote window buffer of a new epoch	Resolved
[L-08] Asymmetric gauge unregistration can result in misallocation of tokens	Resolved

[L-09] Gauge removal before rewards distribution should be explicitly prevented to avoid erroneous loss of vote weight	Acknowledged
[L-10] `DistributionManagerSetup::canDistribute` should return false rather than reverting	Resolved
[L-11] Arbitrary controller addresses can be passed to `DistributionManager::setModuleForController` without validation of the corresponding gauge addresses	Acknowledged
[I-1] `MIN_SPEED` and `DEFAULT_MIN_SPEED` should be merged into a shared constants library	Resolved
[I-2] Unsafe transfer of non-standard ERC-20 tokens	Acknowledged
[I-3] `UniquenessTests` incorrectly refers to the caller being included in the call identifier computation	Resolved
[I-4] Votes will continue to be automatically recast for deactivated gauges if prior voters do not reset	Acknowledged
[I-5] Speed calculation precision handling is ineffective	Resolved
[G-1] Avoid initializing variables to default values	Acknowledged
[G-2] `SpeedCalculator::calcSpeed` should return early if `_moduleBudget` is zero	Resolved
[G-3] Avoid return statements with named return variables	Acknowledged
[G-4] Duplicated validation in `DistributionManager::initialize` can be removed	Resolved
[G-5] Action count increment in `BenqiEcosystemModule::_buildActions` can be simplified	Acknowledged

7 Findings

7.1 Low Risk

7.1.1 Distribution logic can be broken by upgrades to the `GaugeRegistrar` by the DAO to add logic to modify the `enableUpdateVotingPowerHook` configuration

Description: The `DistributionManager` contract hardcodes epoch 0 when reading gauge votes from the `AddressGaugeVoter` contract within `_collectVotes()`:

```
function _collectVotes() internal view returns (...) {
    ...
    for (uint256 j = 0; j < gaugeAddresses.length; j++) {
        address gaugeAddress = gaugeAddresses[j];
        IGaugeRegistrar.RegisteredGauge memory gaugeInfo = gaugeRegistrar.getGaugeInfo(gaugeAddress);

@>      uint256 votes = gaugeVoter.epochGaugeVotes(0, gaugeAddress);

        gaugesByController[i][j] = IDistributionGaugeVote.GaugeVote({
            gaugeAddress: gaugeAddress,
            qiToken: gaugeInfo.qiToken,
            incentive: gaugeInfo.incentive,
            votes: votes
        });
        voteTotals[i] += votes;
    }
}
```

However, `AddressGaugeVoter` conditionally stores votes in either epoch 0 or the current epoch based on the `enableUpdateVotingPowerHook` configuration flag:

```
function _vote(address _account, GaugeVote[] memory _votes) internal {
    ...
    uint256 epoch = getWriteEpochId(); // ← Uses conditional epoch
    AddressVoteData storage voteData = epochVoteData[epoch][_account];
    ...
}

function getWriteEpochId() public view returns (uint256) {
    return enableUpdateVotingPowerHook ? 0 : epochId();
}
```

The system assumes `enableUpdateVotingPowerHook` is always true, storing votes in epoch 0, but:

1. This setting can be changed via calls to `setEnableUpdateVotingPowerHook()` by the `GAUGE_ADMIN_ROLE`.
2. There is no validation during deployment or runtime to enforce this assumption.
3. If set to false, votes are stored in the current epoch (1, 2, 3, ...) but `DistributionManager` always reads from epoch 0.

Regarding calls to `AddressGaugeVoter::setEnableUpdateVotingPowerHook` by the `GAUGE_ADMIN_ROLE`, the `GaugeRegistrar` is granted this permission but the functionality is not currently implemented. However, the `GAUGE_REGISTRAR_ROLE` is granted to the DAO which would allow the implementation to be upgraded and potentially add some logic that violates the assumption that this configuration cannot be changed for BENQI.

Impact: If the `enableUpdateVotingPowerHook` configuration is changed by the DAO, all calls to `distribute()` will erroneously read votes from 0 epoch, resulting in incorrect reward calculations.

Recommended Mitigation: Either explicitly forbid this configuration from being modified or consider modifying the vote collection logic to conditionally retrieve the current epoch if `enableUpdateVotingPowerHook` is false.

BENQI: The `AddressGaugeVoter` should leave the `setEnableVotingPower` enabled as it's the default setting for the Aragon Escrow system. The DAO will be the shared owner, by default of both the address gauge voter and

the gauge registrar, both of which are UUPS. It is correct that this means the registrar could be upgraded and then can make administrative calls to the voter outside the scope of gauge registration as it is the admin. In practice, for BENQI, the DAO will be the same entity so if it's compromised in one place it's compromised in the other anyway. This logic is confirmed in the `DistributionManagerSetup` which reverts if the DAO mismatches between the voter and the `DistributionManager`. Fixed in commit 3ab793e and PR #20.

Cyfrin: Verified. Separation has been added to the operational and administrative `GaugeRegistrar` roles. `DistributionManager::_validateCanDistribute` also now reverts if `GaugeVoter::enableUpdateVotingPowerHook` is disabled.

7.1.2 Missing interface compliance validation in `DistributorManager`

Description: The `DistributionManager` contract does not currently validate that the `budgetAllocator` and controller module contracts implement the required interfaces when set during initialization or via `setBudgetAllocator()` and `setModuleForController()` respectively. Both the `UnifiedBudgetAllocator` and `BenqiCoreModule` contracts implement `ERC165::supportsInterface` to declare interface compliance; however, the setter functions do not verify this before accepting the contracts as valid.

```
// DistributionManager.sol
function _setBudgetAllocator(IBudgetAllocator _budgetAllocator) internal {
    if (address(_budgetAllocator) == address(0)) revert InvalidAddress("budgetAllocator");

    // @audit -  missing interface validation check

    address oldAllocator = address(budgetAllocator);
    budgetAllocator = _budgetAllocator;

    emit BudgetAllocatorSet(oldAllocator, address(_budgetAllocator));
}

function _setModuleForController(
    address _rewardController,
    IDistributionModule _module
) internal {
    if (_rewardController == address(0)) revert InvalidAddress("rewardController");
    if (address(_module) == address(0)) revert InvalidAddress("module");

    // @audit -  missing interface validation check

    if (address(rewardControllerToModule[_rewardController]) != address(0)) {
        revert ControllerAlreadyRegistered(_rewardController);
    }

    _activeRewardControllers.add(_rewardController);
    rewardControllerToModule[_rewardController] = _module;

    emit ModuleForControllerSet(_rewardController, address(_module));
}
```

Impact: Setting an incompatible contract would completely DoS the distribution system.

Recommended Mitigation: Add ERC-165 interface validation checks to both setter functions:

```
if (!IERC165(address(_budgetAllocator)).supportsInterface(type(IBudgetAllocator).interfaceId)) {
      revert InvalidInterface("budgetAllocator");
}
```

BENQI: Acknowledged.

Cyfrin: Acknowledged.

7.1.3 `BenqiEcosystemModule` and `UnifiedBudgetAllocator` are not fully ERC-165 compliant

Description: The `BenqiEcosystemModule` contract inherits both `IDistributionModule` and `IBenqiEcosystemModuleErrors`. According to the ERC-165 specification for interface detection, the `supportsInterface()` function should return `true` for every interface the contract implements. However, the current implementation only checks for `type(IDistributionModule).interfaceId` and the interfaces supported by the parent `ERC165` contract – it fails to include a check for `type(IBenqiEcosystemModuleErrors).interfaceId`.

The `UnifiedBudgetAllocator` contract similarly inherits from multiple interfaces, including `IBudgetAllocator`, `IUnifiedBudgetAllocatorEventsAndErrors`, and `UUPSUpgradeable` which implements `IERC1822Proxiable`. The `supportsInterface()` function is again incomplete since it only returns `true` for `IBudgetAllocator` and `IERC165`, failing to report its support for `IUnifiedBudgetAllocatorEventsAndErrors` and `IERC1822Proxiable`.

Impact: External contracts or off-chain tools that use ERC-165 for interface detection will incorrectly conclude that the `BenqiEcosystemModule` contract does not support the `IBenqiEcosystemModuleErrors` interface and the `UnifiedBudgetAllocator` contract does not support the `IUnifiedBudgetAllocatorEventsAndErrors` or `IERC1822Proxiable` interfaces. This can lead to failed integrations or misbehaving interactions with other systems that rely on proper interface discovery. While the core contract logic remains unaffected, this breaks compliance with the standard and can cause issues with composability and discoverability.

Recommended Mitigation: Modify the `supportsInterface()` functions to include checks for all implemented interfaces.

```
// BenqiEcosystemModule.sol
function supportsInterface(bytes4 _interfaceId) public view virtual override returns (bool) {
    return
        _interfaceId == type(IDistributionModule).interfaceId ||
        _interfaceId == type(IBenqiEcosystemModuleErrors).interfaceId ||
        super.supportsInterface(_interfaceId);
}

// UnifiedBudgetAllocator.sol
function supportsInterface(bytes4 interfaceId) public view virtual override(DaoAuthorizableUpgradeable,
↪  IERC165) returns (bool) {
    return
        interfaceId == type(IBudgetAllocator).interfaceId ||
        interfaceId == type(IUnifiedBudgetAllocatorEventsAndErrors).interfaceId ||
        interfaceId == type(IERC1822Proxiable).interfaceId ||
        interfaceId == type(IERC165).interfaceId ||
}
```

BENQI: Acknowledged.

Cyfrin: Acknowledged.

7.1.4 Maximum number of Aragon DAO actions is not checked within the `DistributorManager`

Description: The `DistributionManager` generates actions for reward distribution but does not currently validate against the DAO's `MAX_ACTIONS` limit before calling `dao.execute()`. The Aragon DAO has a hard limit of 256 actions per execution:

```
// lib/conditions/lib/osx/packages/contracts/src/core/dao/DAO.sol
uint256 internal constant MAX_ACTIONS = 256;

function execute(
        bytes32 _callId,
        Action[] calldata _actions,
        uint256 _allowFailureMap
    )
        external
        override
        nonReentrant
```

```
        auth(EXECUTE_PERMISSION_ID)
        returns (bytes[] memory execResults, uint256 failureMap)
    {
        // Check that the action array length is within bounds.
@>      if (_actions.length > MAX_ACTIONS) {
            revert TooManyActions();
        }
        ...
}
```

But `DistributionManager` calculates the action count without validating against this limit:

```
 function distribute() external auth(DISTRIBUTOR_ROLE) {
    ...
    IExecutor(address(dao())).execute({
        _callId: _generateCallId(),
        _actions: actions, // @audit - no length validation
        _allowFailureMap: 0
    });
    emit RewardsDistributed(currentEpochId);
}
```

Note that there are similarly no limitations on the number of gauges and controllers that are allowed in the system.

Impact: Reaching the action limit could lead to temporary DoS.

Recommended Mitigation: While it is unlikely that this scenario will occur, consider explicitly limiting the number of actions in the `DistributionManager`.

BENQI: Acknowledged.

Cyfrin: Acknowledged.

7.1.5 Budget is always calculated using the full `epochDuration` regardless of the actual timing of distributions

Description: The `DistributionManager` calculates budget allocations based on the full epoch duration and transfers tokens to reward controllers at the end of each epoch. When reward speeds are updated via calls to `setRewardSpeedInternal()`, the budget calculation assumes rewards will be distributed for the entire epoch duration; however, depending on the timing of distribution, this can result in a mismatch between the tokens transferred and tokens actually distributed as rewards.

```
// BenqiCoreModule::calculateMarketBudgetUsed
function calculateMarketBudgetUsed(
    MarketVotes memory _market,
    uint256 _supplySpeed,
    uint256 _borrowSpeed,
    uint256 _epochDuration
) public pure returns (uint256) {
    uint256 budget = 0;

    // Only count budget for registered gauges
    if (_market.hasRegisteredSupplyGauge) {
@>      budget += _supplySpeed * _epochDuration; // @audit - uses full epoch duration
    }

    if (_market.hasRegisteredBorrowGauge) {
@>      budget += _borrowSpeed * _epochDuration; // @audit - uses full epoch duration
    }

    return budget;
}
```

```
// DistributionManager::_buildActions
uint256 amountWithBuffer = _applyBuffer(budgetUsed);
if (amountWithBuffer != 0) {
    actions[actionCount++] = Action({
        to: address(rewardToken),
        value: 0,
        data: abi.encodeWithSelector(
            IERC20.transfer.selector,
            controller,
@>          amountWithBuffer // @audit - full epoch budget transferred
        )
    });
}
```

When speeds are set in the Comptroller, rewards accrue from the instantaneous timestamp onwards while the previous speeds are overwritten and undistributed budget is not taken into consideration:

```
    function setRewardSpeedInternal(uint8 rewardType, QiToken qiToken, uint newSupplyRewardSpeed, uint
    ↪  newBorrowRewardSpeed) internal {
        uint currentSupplyRewardSpeed = supplyRewardSpeeds[rewardType][address(qiToken)];
        uint currentBorrowRewardSpeed = borrowRewardSpeeds[rewardType][address(qiToken)];

        if (currentSupplyRewardSpeed != 0) {
@>          updateRewardSupplyIndex(rewardType, address(qiToken));
        } else if (newSupplyRewardSpeed != 0) {
            Market storage market = markets[address(qiToken)];
            require(market.isListed, "Market is not listed");

            ...
         }

        ...

        if (currentSupplyRewardSpeed != newSupplyRewardSpeed) {
@>          supplyRewardSpeeds[rewardType][address(qiToken)] = newSupplyRewardSpeed;
            emit SupplyRewardSpeedUpdated(rewardType, qiToken, newSupplyRewardSpeed);
        }

       ...
    }

    // lib/BENQI-Smart-Contracts/lending/Comptroller.sol:1112-1130
    function updateRewardSupplyIndex(uint8 rewardType, address qiToken) internal {
        require(rewardType <= 1, "rewardType is invalid");
        RewardMarketState storage supplyState = rewardSupplyState[rewardType][qiToken];
        uint supplySpeed = supplyRewardSpeeds[rewardType][qiToken];
        uint blockTimestamp = getBlockTimestamp();
@>      uint deltaTimestamps = sub_(blockTimestamp, uint(supplyState.timestamp)); // @audit - accrues
↪   from last update
        if (deltaTimestamps > 0 && supplySpeed > 0) {
            uint supplyTokens = QiToken(qiToken).totalSupply();
@>          uint qiAccrued = mul_(deltaTimestamps, supplySpeed); // @audit - deltaTimestamps <
↪   epochDuration
            Double memory ratio = supplyTokens > 0 ? fraction(qiAccrued, supplyTokens) :
            ↪  Double({mantissa: 0});
            Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
            rewardSupplyState[rewardType][qiToken] = RewardMarketState({
                index: safe224(index.mantissa, "new index exceeds 224 bits"),
                timestamp: safe32(blockTimestamp, "block timestamp exceeds 32 bits")
            });
        } else if (deltaTimestamps > 0) {
            supplyState.timestamp = safe32(blockTimestamp, "block timestamp exceeds 32 bits");
        }
```

```
    }
```

Taking in consideration the fact that there is also an additional buffer amount that is transferred to the reward controller and that this discrepancy can occur on every epoch, these non-negligible losses will accumulate over time.

It is understood that there can be no retroactive distributions in the event of delays, and the addition of a voting buffer is added to account for potential delays based on the fact that, while operators are assumed to trigger distributions at roughly the same interval, this is not currently fully automated and thus not guaranteed. However, it is still not clear when exactly the operators are expected to execute distribution at roughly the same time every epoch strictly before the epoch ends. If so, this means that a portion of the budget will always be wasted, exacerbated by the addition of the buffer amount.

This is unavoidable unless perhaps distributions can occur after the end of the epoch, e.g. in the hour window before voting starts again in the next epoch, in which case the operators should optimally target strictly the end of the epoch or later. Unfortunately, as documented separately, this is not possible based on the current logic as an attempted distribution during this period would incorrectly mark the next distribution as complete even when voting has not yet started. Ideally, this logic would be modified to support distributions for epoch N during the initial buffer period of epoch N+1.

Impact: Depending on when `distribute()` is called, a part of the transferred budget may never be distributed.

Proof of Concept: Consider the following example:

- Day 0-7: Voting active
- Day 7: Voting ends
- Day 10: `distribute()` is called (3 days after voting ends)
  - Budget calculated: speed * 2 weeks = 1000 tokens
  - Tokens transferred to Comptroller: 1000 tokens (not counting extra buffer)
  - New speed set: 500 tokens/week (71.43 tokens/day)
- Day 10-14: Rewards accrue at 71.43 tokens/day = ~286 tokens
- Day 14: Epoch N ends (but speed continues unchanged)
- Day 14-21 (Epoch N+1): Voting active, rewards STILL accrue at 71.43 tokens/day = ~500 tokens
- Day 21: Voting ends
- Day 22: Next `distribute()` is called
- Total time at old speed: Day 10 to Day 22 = 12 days
- Total distributed from Epoch N budget: 71.43 * 12 = ~857 tokens
- Wasted from Epoch N: 1000 - 857 = ~143 tokens

The following test should be added to `DistributeManager.distribute.t,sol`:

```
function test_wastedBudgetPoC() public {
    // Setup two gauges
    vm.startPrank(ADMIN);
    address gauge1 = gaugeRegistrar.registerGauge(
        QI_TOKEN_1, IGaugeRegistrar.Incentive.Supply, address(coreComptroller), "Gauge 1"
    );
    address gauge2 = gaugeRegistrar.registerGauge(
        QI_TOKEN_2, IGaugeRegistrar.Incentive.Supply, address(coreComptroller), "Gauge 2"
    );
    vm.stopPrank();

    coreComptroller.setMarketListed(QI_TOKEN_1, true);
    coreComptroller.setMarketListed(QI_TOKEN_2, true);
```

```
    // Epoch 0: gauge1 gets 80% of votes, gauge2 gets 20%
    addressGaugeVoter.setEpochGaugeVotes(0, gauge1, 800 ether);
    addressGaugeVoter.setEpochGaugeVotes(0, gauge2, 200 ether);

    // Epoch 0: Voting ends Day 7, distribute() called Day 10 (3 day delay)
    uint256 distributeDay10 = 1 weeks + 3 days;
    vm.warp(distributeDay10);

    vm.prank(DISTRIBUTOR);
    distributionManager.distribute();

    uint256 gauge1FirstSpeed = coreComptroller.supplyRewardSpeeds(0, QI_TOKEN_1);

    // Epoch 1: Votes SHIFT - gauge1 now only gets 30%, gauge2 gets 70%
    addressGaugeVoter.setEpochGaugeVotes(0, gauge1, 300 ether);
    addressGaugeVoter.setEpochGaugeVotes(0, gauge2, 700 ether);

    uint256 distributeDay22 = 2 weeks + 1 weeks + 1 days;
    vm.warp(distributeDay22);

    vm.prank(DISTRIBUTOR);
    distributionManager.distribute(); // Overwrites speeds from Epoch 0

    // Check: gauge1 speed REDUCED (now only 30% of budget)
    uint256 gauge1NewSpeed = coreComptroller.supplyRewardSpeeds(0, QI_TOKEN_1);

    assertGt(gauge1FirstSpeed, gauge1NewSpeed, "Speed REDUCED when votes decreased and budget wasted!");
}
```

Recommended Mitigation: Distributions for epoch N would ideally be made during the initial buffer period of epoch N+1.

The integration tests notably do not cover distributions beyond the expected token transfers to the distributor rather than also testing claims by market participants. To effectively demonstrate the scenario described above, an assertion should be implemented for the claimable vs transferred amount.

BENQI: We acknowledge that the nature of the distributor means that, indeed some funds in the contracts may go unclaimed due to the buffer.

We also note that each rewards controller implementation defines sweep/rescue functions for the admin to recover funds if they require it. These are `_grantQi` and `_rescueFunds` for the comptroller and distributor respectively.

Our idea was that a small excess is preferable to running short, as funds can be asynchronously retrieved without affecting users, while the reverse is not true.

We also note that the admins are freely able to reduce the buffer, in order to reduce the excess.

Cyfrin: Acknowledged.

7.1.6 Single `rewardToken` design is incompatible with multi-reward emission support

Description: `BenqiEcosystemModule` assumes QI as the single immutable `rewardToken` always referenced directly within `_processGauge()`:

```
 function _processGauge(
    IMultiRewardDistributor _distributor,
    GaugeVote memory _gauge,
    uint256 _totalVotes,
    uint256 _budget,
    uint256 _epochDuration,
    uint256 _emissionCap
) internal view returns (ProcessGaugeReturnValue memory returnData) {
    ...
```

```
    if (_gauge.incentive == IGaugeRegistrar.Incentive.Supply) {
        returnData.actionData = abi.encodeCall(
            IMultiRewardDistributor._updateSupplySpeed,
@>          (QiToken(_gauge.qiToken), rewardToken, newSpeed)
        );
    } else {
        returnData.actionData = abi.encodeCall(
            IMultiRewardDistributor._updateBorrowSpeed,
@>          (QiToken(_gauge.qiToken), rewardToken, newSpeed)
        );
    }
}
```

However, `MultiRewardDistributor` supports multiple reward tokens per `qiToken` market:

```
/// @notice The main data storage for this contract, holds a mapping of qiToken to array
//          of market configs
mapping(address => MarketEmissionConfig[]) public marketConfigs;
```

> Each market has an array of configs, each with a unique emission token owned by a specific team/user. That owner can adjust supply and borrow emissions, end times, and ...

This can also be observed in the logic of `_updateSupplySpeed()` and `_updateBorrowSpeed()`, which are the calls encoded within the `BenqiEcosystemModule`:

```
    function _updateSupplySpeed(
        QiToken _qiToken,
        address _emissionToken,
        uint256 _newSupplySpeed
    ) external onlyEmissionConfigOwnerOrAdmin(_qiToken, _emissionToken) {
        MarketEmissionConfig storage emissionConfig = fetchConfigByEmissionToken(
@>          _qiToken,
@>          _emissionToken
        );
        ...
    }
```

```
function fetchConfigByEmissionToken(
    QiToken _qiToken,
    address _emissionToken
) internal view returns (MarketEmissionConfig storage) {
    MarketEmissionConfig[] storage configs = marketConfigs[address(_qiToken)];
    for (uint256 index = 0; index < configs.length; index++) {
        MarketEmissionConfig storage emissionConfig = configs[index];
        if (emissionConfig.config.emissionToken == _emissionToken) {
            return emissionConfig;
        }
    }

    revert("Unable to find emission token in qiToken configs");
}
```

Impact: The `BenqiEcosystemModule` can only adjust the speed for whichever single token was set in the constructor. It cannot differentiate or control the other emission tokens, even though they can exist and be active in the `MultiRewardDistributor`.

Recommended Mitigation: Consider allowing the `BenqiEcosystemModule` and the `Distributor` to pass different reward tokens.

BENQI: Acknowledged, this is purely a QI distributor.

Cyfrin: Acknowledged.

7.1.7 Reward distribution can be skipped when executed within the vote window buffer of a new epoch

Description: `DistributionManager::distribute` is validated to only be callable when voting is not active. This is achieved by calling `_validateCanDistribute()`; however, this fails to account for the `VOTE_WINDOW_BUFFER` period at the start of each epoch.

The Clock contract defines epochs with the following structure:

- `EPOCH_DURATION`: 2 weeks
- `VOTE_DURATION`: 1 week
- `VOTE_WINDOW_BUFFER`: 1 hour

```
    function _validateCanDistribute(IClock _clock) internal view {
        if (_activeRewardControllers.length() == 0) revert NoModulesConfigured();
@>      if (_clock.votingActive()) revert VotingStillActive(); // @audit - returns false in buffer
↪   period

@>      uint256 currentEpochId = _clock.currentEpoch(); // @audit - returns N+1
@>      if (_isEpochDistributed(currentEpochId)) { // @audit - checks if N+1 distributed, not N
            revert EpochAlreadyDistributed(currentEpochId);
        }
    }
```

For each epoch, voting is only active from `T = 1 hour` to `T = 1 week - 1 hour`; however, when a new epoch begins, there's a 1-hour buffer period where:

- `votingActive()` returns false
- `currentEpoch()` returns the new epoch N+1

This incorrectly allows for distributions to occur in the first hour of the epoch N+1, preventing subsequent distribution for the epoch.

Impact: If `distribute()` is called during the 1-hour window at the start of epoch N+1, for example in the case of late distribution of epoch N, both epochs N and N+1 will never be distributed. The system will mark epoch N+1 as distributed before any votes have been cast for it.

Proof of Concept: The following test should be added to `DistributorManager.distribute.t.sol`:

```
function testPoC_bufferWindowSkipsEpoch() public {
    // Setup gauge and votes for epoch 0
    vm.prank(ADMIN);
    address gauge = gaugeRegistrar.registerGauge(
        QI_TOKEN_1, IGaugeRegistrar.Incentive.Supply, address(coreComptroller), "Gauge"
    );
    coreComptroller.setMarketListed(QI_TOKEN_1, true);
    addressGaugeVoter.setEpochGaugeVotes(0, gauge, 1000 ether);

    // Warp to 30 minutes into epoch 1 (in VOTE_WINDOW_BUFFER)
    vm.warp(2 weeks + 30 minutes);

    // Verify preconditions: epoch 1, voting inactive, in buffer period
    assertEq(clock.currentEpoch(), 1);
    assertFalse(clock.votingActive());
    assertLt(clock.elapsedInEpoch(), 1 hours);

    // Call distribute() - should fail but doesn't
    vm.prank(DISTRIBUTOR);
    distributionManager.distribute();

    // BUG: Epoch 1 marked distributed, Epoch 0 skipped forever
    assertFalse(distributionManager.isDistributed(0), "Epoch 0 never distributed!");
    assertTrue(distributionManager.isDistributed(1), "Epoch 1 wrongly marked distributed");
```

```
}
```

As described, the epoch 0 rewards were never distributed and the epoch 1 was distributed with 0 votes and zero budget.

Recommended Mitigation: Either explicitly prevent distribution during the vote window buffer or, preferably, allow distribution for the previous epoch within this period.

BENQI: Fixed in commit PR #13.

Cyfrin: Verified. Distributions can no longer be made during the pre-vote buffer period.

7.1.8 Asymmetric gauge unregistration can result in misallocation of tokens

Description: The reward speed preservation logic in the `BenqiCoreModule` contract enables indefinite QI token distributions for markets associated with asymmetrically unregistered or otherwise expired supply/borrow gauges. Specifically, when a gauge is unregistered via `GaugeRegistrar::unregisterGauge`, it is removed from the internal tracking sets `_allGauges` and `_gaugesByRewardController` before being deactivated in the linked `AddressGaugeVoter`, preventing future votes. However, in `BenqiCoreModule::generateActions`, the `calculateSpeeds()` function checks the `hasRegisteredSupplyGauge` and `hasRegisteredBorrowGauge` flags. If a flag is false, the existing supply or borrow reward speed from `CoreComptroller` is preserved rather than being set to zero. This intentional design avoids abrupt changes to reward speed but lacks an automatic zeroing mechanism, allowing speeds to persist at their last non-zero value unless governance manually intervenes with a direct call to `setRewardSpeed()`.

Impact: Markets with asymmetrically unregistered gauges can continue distributing QI rewards indefinitely, leading to misallocation of tokens outside the system's budgeted epochs. This could exhaust QI reward pools over time or necessitate unplanned replenishments. The economic loss scales with market activity (e.g., supply/borrow volumes triggering claims), preserved speed levels, and duration before detection. It introduces an operational risk reliant on vigilant governance; if unregistration occurs without follow-up (e.g., due to oversight in a DAO process), it may result in significant, cumulative token leakage, though this is not directly exploitable by external actors.

Proof of Concept: The following test should be added to `BenqiCoreModule.generateActions.t.sol`:

```
function test_deactivateSupplyGauge_usesExistingSpeed() public {
    // Set existing speeds
    comptroller.setRewardSpeeds(QITOKEN_1, 1000, 2000);

    // Only provide borrow gauge, not supply
    IDistributionGaugeVote.GaugeVote[] memory gauges = new IDistributionGaugeVote.GaugeVote[](1);
    gauges[0] = IDistributionGaugeVote.GaugeVote({
        gaugeAddress: GAUGE_1,
        qiToken: QITOKEN_1,
        incentive: IGaugeRegistrar.Incentive.Borrow,
        votes: 100
    });

    (Action[] memory actions, ) = module.generateActions(
        address(comptroller),
        gauges,
        MODULE_BUDGET,
        EPOCH_DURATION
    );

    // Decode the action to check speeds
    (, , uint256 supplySpeed, uint256 borrowSpeed) = decodeSetRewardSpeed(actions[0].data);

    // Borrow speed should be calculated from votes
    uint256 expectedBorrowSpeed = ((MODULE_BUDGET / EPOCH_DURATION) * 100) / 100;
    assertEq(borrowSpeed, expectedBorrowSpeed);
    assertGt(borrowSpeed, 0);
    assertNotEq(borrowSpeed, 2000);
```

```
    // Supply speed should be preserved
    assertEq(supplySpeed, 1000);
}
```

Recommended Mitigation: Consider removing the speed preservation logic. This could be achieved by modifying `BenqiCoreModule::calculateSpeeds` to always recalculate speeds for all gauges based on current votes, setting them to zero or `MIN_SPEED` when `hasRegisteredSupplyGauge` or `hasRegisteredBorrowGauge` is false, rather than preserving the current comptroller speeds.

BENQI: This turned out to be a bigger PR than anticipated. We first implemented the unregistration hook via a pattern of:

Registrar calls Distribution Manager via a dedicated endpointed auth'd to the Registrar. The Distribution Manager queries the associated module to get the unregister actions. The Distribution manager executes as it has the permission on the DAO for the speed adjustments.

This created a problem though: gauges could be registered without a check to see if they actually existed. This meant gauges could be registered but could not be unregistered and this would permanently block distribution as, in both cases, we would attempt to read from a nonlisted gauge.

We therefore added 2 safety measures:

1. A check to see the gauge exists, during registration. This should cover 95% of cases.
2. A try/catch with a dedicated event emitted if unregistration fails. This would necessitate further action from the admin to understand the issue but is the same as the base case.

Fixed in PR #24.

Cyfrin: Verified. Unregistration of a valid gauge first updates the current speed on the `Comptroller` to the `MIN_SPEED` in the unregister actions such that future allocations are minimized.

7.1.9 Gauge removal before rewards distribution should be explicitly prevented to avoid erroneous loss of vote weight

Description: Reward distributions occur following the conclusion of the voting period, in which Miles token holders allocate their voting power toward their choses gauge(s). However, if a gauge is removed after the voting period concludes but before the distribution phase begins, no rewards will be distributed for that gauge. Consequently, all votes cast for the removed gauge will be rendered obsolete.

Impact: All votes cast during the affected epoch for the removed gauge will be invalidated, leading to a total loss of voting weight and corresponding rewards for the voters.

Recommended Mitigation: `DistributionManager::_isEpochDistributed` determines whether an epoch has already been distributed. This check can be combined with the condition that voting has not yet started to create a new method within the `DistributionManager`. This new method can then be utilized within the `GaugeRegistrar` to ensure that gauges cannot be removed during this critical interval, preventing loss of user votes and ensuring proper reward distribution.

BENQI: Acknowledged. The solution to this is quite complex and not worth.

One way would be that we only allow unregistering gauge when time is between epoch start and voting start, but this would mean that if admin wants to remove gauge, he has to wait for a very specific time to be able to do so.

Cyfrin: Acknowledged.

7.1.10 `DistributionManagerSetup::canDistribute` should return false rather than reverting

Description: `DistributionManagerSetup::canDistribute` currently returns true if distribution can proceed but reverts from within `_validateCanDistribute()` if it cannot:

```
    function canDistribute() public view returns (bool) {
        IClock clock = IClock(gaugeVoter.clock());
@>      _validateCanDistribute(clock);
        return true;
    }

    function _validateCanDistribute(IClock _clock) internal view {
@>      if (_activeRewardControllers.length() == 0) revert NoModulesConfigured();
@>      if (_clock.votingActive()) revert VotingStillActive();

        uint256 currentEpochId = _clock.currentEpoch();
        if (_isEpochDistributed(currentEpochId)) {
@>          revert EpochAlreadyDistributed(currentEpochId);
        }
    }
```

Rather than reverting, this view function should return false so it can be reliably called by consumers.

BENQI: Fixed in PR #20.

Cyfrin: Verified. A call to this function now either reverts or succeeds with empty return data.

7.1.11 Arbitrary controller addresses can be passed to `DistributionManager::setModuleForController` without validation of the corresponding gauge addresses

Description: The DAO can currently pass any arbitrary controller address in calls to `DistributionManager::setModuleForController`; however, this logic should additionally validate that the gauges of the controller are registered on the `GaugeRegistrar` and `AddressGaugeVoter` to avoid adding an invalid controller to the `_activeRewardControllers` set:

```
/// @dev Internal function to set module for controller
function _setModuleForController(
    address _rewardController,
    IDistributionModule _module
) internal {
    if (_rewardController == address(0)) revert InvalidAddress("rewardController");
    if (address(_module) == address(0)) revert InvalidAddress("module");

    // Enforce one module per controller constraint
    if (address(rewardControllerToModule[_rewardController]) != address(0)) {
        revert ControllerAlreadyRegistered(_rewardController);
    }

@>  _activeRewardControllers.add(_rewardController);

    rewardControllerToModule[_rewardController] = _module;

    emit ModuleForControllerSet(_rewardController, address(_module));
}
```

Impact is low since the DAO is unlikely to deliberately configure this incorrectly. Nevertheless, `_validateCanDistribute()` can return true in this scenario since it only checks the length of active reward controllers is non-zero rather than the gauges corresponding to those controllers:

```
    function _validateCanDistribute(IClock _clock) internal view {
@>      if (_activeRewardControllers.length() == 0) revert NoModulesConfigured();
        if (_clock.votingActive()) revert VotingStillActive();

        uint256 currentEpochId = _clock.currentEpoch();
        if (_isEpochDistributed(currentEpochId)) {
            revert EpochAlreadyDistributed(currentEpochId);
```

```
        }
    }
```

Given that within `_collectVotes()` the gauge addresses are attempted to be queried from the `GaugeRegistrar`, an empty array will always be returned for invalid controller addresses:

```
address controller = _activeRewardControllers.at(i);
address[] memory gaugeAddresses = gaugeRegistrar.getGaugesByRewardController(
    controller
);
```

Furthermore, execution will only revert in `UnifiedBudgetAllocator::allocateBudget` due to the total collected votes being zero if there are no valid registered gauges across any of the active controllers:

```
// Calculate total votes across all controllers
uint256 totalVotes = 0;
for (uint256 i = 0; i < _voteTotals.length; i++) {
    totalVotes += _voteTotals[i];
}

// Revert if no votes to prevent division by zero
if (totalVotes == 0) revert ZeroTotalVotes();
```

Thus, if the DAO makes a mistake when calling `setModuleForController()` with a controller that is not already associated with gauge(s) registered on the `GaugeRegistrar`, distribution will be skipped and rewards will not be sent to the expected gauge.

Recommended Mitigation: Consider validating that the gauge addresses associated with a given controller are already registered on the `GaugeRegistrar`.

BENQI: Acknowledged.

Cyfrin: Acknowledged.

7.2 Informational

7.2.1 `MIN_SPEED` and `DEFAULT_MIN_SPEED` should be merged into a shared constants library

Description: The `DistributionManager`, `BenqiCoreModule`, and `BenqiEcosystemModule` contracts each independently define a constant for the minimum reward speed, `DEFAULT_MIN_SPEED` and `MIN_SPEED` respectively. Although they currently share the same value of 1 wei purpose, to ensure a minimal reward flow when gauge votes drop to zero, they are not linked.

The `DistributionManager` uses its `DEFAULT_MIN_SPEED` to calculate the maximum permissible budget overrun. In contrast, the modules use their own private `MIN_SPEED` constants to determine the actual budget required, which includes applying this minimum speed floor.

Impact: This duplication adds additional overheads introduces the risk of the values becoming inconsistent during future development. If `MIN_SPEED` in a module is changed to a value that differs from `DEFAULT_MIN_SPEED` in the `DistributionManager`, the values will no longer be equal.

Recommended Mitigation: To ensure consistency and simplify maintenance, the minimum speed constant should be defined in a single location:

1. Create a new library (e.g., `Constants.sol`) to hold shared constants.
2. Define a single `MIN_SPEED` constant in this new library.
3. Remove the separate `DEFAULT_MIN_SPEED` and `MIN_SPEED` definitions from `DistributionManager`, `BenqiCoreModule`, and `BenqiEcosystemModule`.
4. Have all three contracts import the new constants library and reference the single `MIN_SPEED` value. This ensures that all components of the system are always operating with the same parameter, preventing potential inconsistencies.

BENQI: Fixed in PR #29.

Cyfrin: Verified.

7.2.2 Unsafe transfer of non-standard ERC-20 tokens

Description: The `DistributionManager` uses the standard ERC-20 `transfer()` function instead of a safe transfer equivalent when distributing reward tokens to controllers. While the reward token (QI) is a known, trusted contract that conforms to the ERC-20 specification, direct use of `transfer()` can lead to silent failures if this is not the case and the token does not correctly conform to the standard.

```
// DistributionManager.sol - Lines 416-424
actions[actionCount++] = Action({
    to: address(rewardToken),
    value: 0,
    data: abi.encodeWithSelector(
        IERC20.transfer.selector, // @audit - uses transfer, not safeTransfer
        controller,
        amountWithBuffer
    )
});
```

The standard `IERC20::transfer` may:

- Return false on failure without reverting (e.g. USDT).
- Not return any value at all (e.g. BNB)

In both cases, the transaction would continue executing, the epoch would be marked as distributed, but the controller would not receive the tokens. This creates an inconsistent state where distribution appears successful but tokens were never transferred.

Recommended Mitigation: Consider using a checked safe transfer equivalent instead of direct ERC-20 transfer.

BENQI: Acknowledged.

Cyfrin: Acknowledged.

7.2.3 `UniquenessTests` incorrectly refers to the caller being included in the call identifier computation

Description: `UniquenessTests::test_CallIdIncludesCaller` is implemented based on the assumption that the call identifier will differ when called by two different addresses as the caller is included in the computation; however, this is incorrect and the identifiers differ only because the nonce is incremented between `generateCallId()` invocations:

```
function test_CallIdIncludesCaller() public {
    vm.prank(user1);
    bytes32 id1 = callerId.generateCallId();

    vm.prank(user2);
    bytes32 id2 = callerId.generateCallId();

    assertTrue(id1 != id2);
}
```

Recommended Mitigation: Remove this test altogether, or modify it to use snapshots to validate that, for the same nonce, the returned identifiers will be equal when called by distinct addresses.

BENQI: Fixed in PR #28.

Cyfrin: Verified.

7.2.4 Votes will continue to be automatically recast for deactivated gauges if prior voters do not reset

Description: The `test_calculateSpeeds_positiveVotesNoGauge()` function in `BenqiCoreModule.calculateSpeeds.t.sol` contains the assumption that such a scenario in which a non-existent gauge has non-zero votes should not happen in practice:

```
function test_calculateSpeeds_positiveVotesNoGauge() public view {
    // This shouldn't happen in practice, but testing edge case
    IBenqiCoreState.MarketVotes memory market = IBenqiCoreState.MarketVotes({
        qiToken: QITOKEN_1,
        supplyVotes: 100, // Has votes but no gauge
        borrowVotes: 200,
        hasRegisteredSupplyGauge: false,
        hasRegisteredBorrowGauge: false
    });

    uint256 currentSupplySpeed = 5000;
    uint256 currentBorrowSpeed = 6000;

    (uint256 supplySpeed, uint256 borrowSpeed) = module.calculateSpeeds(
        market,
        1000, // totalVotes
        MODULE_BUDGET,
        EPOCH_DURATION,
        currentSupplySpeed,
        currentBorrowSpeed
    );

    // No gauges, should preserve current speeds despite votes
    assertEq(supplySpeed, currentSupplySpeed);
    assertEq(borrowSpeed, currentBorrowSpeed);
}
```

However, this is not strictly true and can occur for deactivated gauges because votes are automatically recast in the absence of a reset when previous voters vote for other gauges. This could result in an unregistered/deactivated gauge having phantom voting power, for example if the gauge is deactivated directly on the `AddressGaugeVoter` without also being unregistered from the `GaugeRegistrar` since the gauges for the controller are retrieved from the `GaugeRegistrar`, meaning this deactivation this won't be reflected. The auto-recast votes will not be reset, so any controller that has its gauge deactivated in this manner will continue to receive a share of rewards.

Impact: Distributions are unaffected by gauge deactivations in `AddressGaugeVoter`.

Proof of Concept: The following test should be added to `BenqiIntegrationSingleEpoch.t.sol`:

```
function test_votesRecastAndRewardsDistributedToDeactivatedGauge() public {
    if (!useFork) return;

    // Setup budget for multiple epochs
    {
        uint budget = allocator.totalQiBudget();
        uint totalNeeded = (budget + ((budget * BUFFER) / 10_000)) * 2; // For 2 epochs
        mintOrSend(address(dao), totalNeeded, useFork);
    }

    // Ensure we're not in a voting window before registering gauges
    vm.warp(clock.epochVoteEndTs() + 1);
    assertFalse(voter.votingActive(), "Should not be voting active");

    address gEcoSupply;
    address gCoreBorrow;
    vm.startPrank(address(dao));
    {
        gEcoSupply = registrar.registerGauge(
            address(qiTokenM0),
            IGaugeRegistrar.Incentive.Supply,
            address(distributor),
            "USDC Ecosystem Supply Gauge"
        );

        gCoreBorrow = registrar.registerGauge(
            address(qiTokenC0),
            IGaugeRegistrar.Incentive.Borrow,
            address(coreComptroller),
            "USDC Core Borrow Gauge"
        );
    }
    vm.stopPrank();

    // define a user
    address guy = address(0x1337);

    // mint them tokens
    mintOrSend(guy, 1000e18, useFork);

    // stake the tokens
    uint tokenId1;
    uint tokenId2;
    vm.startPrank(guy);
    {
        miles.approve(address(escrow), 1000e18);
        tokenId1 = escrow.createLock(990e18);
        tokenId2 = escrow.createLock(10e18);
        adapter.delegate(guy);
        assertEq(adapter.getVotes(guy), 1000e18, "votes");
    }
    vm.stopPrank();
```

```
// go to voting window
vm.warp(clock.epochVoteStartTs());
assertTrue(voter.votingActive(), "V!A");

// vote - 50/50 split between ecosystem and core
GaugeVote[] memory votes = new GaugeVote[](2);
votes[0] = GaugeVote({gauge: gEcoSupply, weight: 10000});
votes[1] = GaugeVote({gauge: gCoreBorrow, weight: 10000});

assertEq(adapter.getVotes(guy), 1000e18, "votes");

vm.prank(guy);
voter.vote(votes);

assertEq(voter.epochGaugeVotes(0, gEcoSupply), 500e18, "eco gauge votes");
assertEq(voter.epochGaugeVotes(0, gCoreBorrow), 500e18, "core gauge votes");

// go to dist window
vm.warp(clock.epochNextCheckpointTs());
assertFalse(voter.votingActive(), "VA");

// Capture state before distribution
DualDistributionState memory state;
{
    state.currentEpochId = clock.currentEpoch();
    state.initialDaoBalance = miles.balanceOf(address(dao));
    state.initialDistributorBalance = miles.balanceOf(address(distributor));
    state.initialCoreBalance = miles.balanceOf(address(coreComptroller));

    // Capture previous epoch distribution status
    if (state.currentEpochId > 0) {
        state.prevEpochDistributed = dmgr.isDistributed(state.currentEpochId - 1);
    }

    // Ecosystem state
    state.ecoConfigBefore = distributor.getConfigForMarket(qiTokenM0, address(miles));

    // Core state
    state.coreSupplySpeedBefore = coreComptroller.supplyRewardSpeeds(0, address(qiTokenC0));
    state.coreBorrowSpeedBefore = coreComptroller.borrowRewardSpeeds(0, address(qiTokenC0));

    // Calculate expected values
    state.totalBudget = allocator.totalQiBudget();
    // Since we have 2 controllers and votes are split 50/50
    // First allocator splits budget between controllers based on total votes
    // In this case, each controller gets 50% of total budget
    state.ecosystemBudget = state.totalBudget / 2;
    state.coreBudget = state.totalBudget / 2;

    // Each gauge gets 100% of its controller's budget (since only one gauge per controller)
    state.expectedEcoSpeed = state.ecosystemBudget / clock.epochDuration();
    state.expectedCoreSpeed = state.coreBudget / clock.epochDuration();

    // Calculate actual transfers based on budgetUsed (speed * epochDuration)
    // This accounts for rounding in integer division
    uint256 ecoBudgetUsed = state.expectedEcoSpeed * clock.epochDuration();
    uint256 coreBudgetUsed = state.expectedCoreSpeed * clock.epochDuration();

    // DistributionManager transfers budgetUsed + buffer, NOT allocatedBudget + buffer
    state.expectedEcoTransfer = ecoBudgetUsed + ((ecoBudgetUsed * BUFFER) / 10_000);
    state.expectedCoreTransfer = coreBudgetUsed + ((coreBudgetUsed * BUFFER) / 10_000);
    state.expectedTotalTransfer = state.expectedEcoTransfer + state.expectedCoreTransfer;
```

```
    }

    // distribute
    vm.prank(address(dao));
    dmgr.distribute();

    // Capture post-distribution state
    {
        state.ecoConfigAfter = distributor.getConfigForMarket(qiTokenM0, address(miles));
        state.coreSupplySpeedAfter = coreComptroller.supplyRewardSpeeds(0, address(qiTokenC0));
        state.coreBorrowSpeedAfter = coreComptroller.borrowRewardSpeeds(0, address(qiTokenC0));
    }

    // Run core assertions
    {
        // 1. Distribution marked for current epoch only
        assertTrue(
            dmgr.isDistributed(state.currentEpochId),
            "Current epoch should be marked as distributed"
        );

        // Verify previous epoch status unchanged (if exists)
        if (state.currentEpochId > 0) {
            assertEq(
                dmgr.isDistributed(state.currentEpochId - 1),
                state.prevEpochDistributed,
                "Previous epoch distribution status should remain unchanged"
            );
        }

        // Verify next epoch is NOT distributed
        assertFalse(
            dmgr.isDistributed(state.currentEpochId + 1),
            "Next epoch should not be distributed"
        );

        // 2. Token balances updated
        assertEq(
            miles.balanceOf(address(dao)),
            state.initialDaoBalance - state.expectedTotalTransfer,
            "DAO balance should decrease by total actual transfer"
        );

        assertEq(
            miles.balanceOf(address(distributor)),
            state.initialDistributorBalance + state.expectedEcoTransfer,
            "Distributor should receive ecosystem transfer"
        );

        assertEq(
            miles.balanceOf(address(coreComptroller)),
            state.initialCoreBalance + state.expectedCoreTransfer,
            "Core comptroller should receive core transfer"
        );

        // 3. Ecosystem market speeds updated
        assertEq(
            state.ecoConfigAfter.supplyEmissionsPerSec,
            state.expectedEcoSpeed,
            "Ecosystem supply speed should match expected"
        );
        assertEq(
            state.ecoConfigAfter.borrowEmissionsPerSec,
```

```
            state.ecoConfigBefore.borrowEmissionsPerSec,
            "Ecosystem borrow speed should remain unchanged (supply gauge only)"
        );

        // 4. Core market speeds updated
        assertEq(
            state.coreSupplySpeedAfter,
            state.coreSupplySpeedBefore,
            "Core supply speed should remain unchanged (borrow gauge only)"
        );
        assertEq(
            state.coreBorrowSpeedAfter,
            state.expectedCoreSpeed,
            "Core borrow speed should match expected"
        );
    }

    // Deactivate gEcoSupply
    vm.prank(address(dao));
    voter.deactivateGauge(gEcoSupply);

    // go to voting window
    vm.warp(clock.epochVoteStartTs());
    assertTrue(voter.votingActive(), "V!A");

    // Votes are automatically recast
    assertEq(voter.epochGaugeVotes(0, gEcoSupply), 500e18, "eco gauge votes");
    assertEq(voter.epochGaugeVotes(0, gCoreBorrow), 500e18, "core gauge votes");

    // go to dist window
    vm.warp(clock.epochNextCheckpointTs());
    assertFalse(voter.votingActive(), "VA");

    // Capture state before distribution
    {
        state.currentEpochId = clock.currentEpoch();
        state.initialDaoBalance = miles.balanceOf(address(dao));
        state.initialDistributorBalance = miles.balanceOf(address(distributor));
        state.initialCoreBalance = miles.balanceOf(address(coreComptroller));

        // Capture previous epoch distribution status
        if (state.currentEpochId > 0) {
            state.prevEpochDistributed = dmgr.isDistributed(state.currentEpochId - 1);
        }

        // Ecosystem state
        state.ecoConfigBefore = distributor.getConfigForMarket(qiTokenM0, address(miles));

        // Core state
        state.coreSupplySpeedBefore = coreComptroller.supplyRewardSpeeds(0, address(qiTokenC0));
        state.coreBorrowSpeedBefore = coreComptroller.borrowRewardSpeeds(0, address(qiTokenC0));

        // Calculate expected values
        state.totalBudget = allocator.totalQiBudget();
        // Since we have 2 controllers and votes are split 50/50
        // First allocator splits budget between controllers based on total votes
        // In this case, each controller gets 50% of total budget
        state.ecosystemBudget = state.totalBudget / 2;
        state.coreBudget = state.totalBudget / 2;

        // Each gauge gets 100% of its controller's budget (since only one gauge per controller)
        state.expectedEcoSpeed = state.ecosystemBudget / clock.epochDuration();
        state.expectedCoreSpeed = state.coreBudget / clock.epochDuration();
```

```
        // Calculate actual transfers based on budgetUsed (speed * epochDuration)
        // This accounts for rounding in integer division
        uint256 ecoBudgetUsed = state.expectedEcoSpeed * clock.epochDuration();
        uint256 coreBudgetUsed = state.expectedCoreSpeed * clock.epochDuration();

        // DistributionManager transfers budgetUsed + buffer, NOT allocatedBudget + buffer
        state.expectedEcoTransfer = ecoBudgetUsed + ((ecoBudgetUsed * BUFFER) / 10_000);
        state.expectedCoreTransfer = coreBudgetUsed + ((coreBudgetUsed * BUFFER) / 10_000);
        state.expectedTotalTransfer = state.expectedEcoTransfer + state.expectedCoreTransfer;
    }

    // distribute
    vm.prank(address(dao));
    dmgr.distribute();

    // Capture post-distribution state
    {
        state.ecoConfigAfter = distributor.getConfigForMarket(qiTokenM0, address(miles));
        state.coreSupplySpeedAfter = coreComptroller.supplyRewardSpeeds(0, address(qiTokenC0));
        state.coreBorrowSpeedAfter = coreComptroller.borrowRewardSpeeds(0, address(qiTokenC0));
    }

    // Run core assertions
    {
        // 1. Distribution marked for current epoch only
        assertTrue(
            dmgr.isDistributed(state.currentEpochId),
            "Current epoch should be marked as distributed"
        );

        // Verify previous epoch status unchanged (if exists)
        if (state.currentEpochId > 0) {
            assertEq(
                dmgr.isDistributed(state.currentEpochId - 1),
                state.prevEpochDistributed,
                "Previous epoch distribution status should remain unchanged"
            );
        }

        // Verify next epoch is NOT distributed
        assertFalse(
            dmgr.isDistributed(state.currentEpochId + 1),
            "Next epoch should not be distributed"
        );

        // 2. Token balances updated
        assertEq(
            miles.balanceOf(address(dao)),
            state.initialDaoBalance - state.expectedTotalTransfer,
            "DAO balance should decrease by total actual transfer"
        );

        assertEq(
            miles.balanceOf(address(distributor)),
            state.initialDistributorBalance + state.expectedEcoTransfer,
            "Distributor should receive ecosystem transfer"
        );

        assertEq(
            miles.balanceOf(address(coreComptroller)),
            state.initialCoreBalance + state.expectedCoreTransfer,
            "Core comptroller should receive core transfer"
```

```
        );

        // 3. Ecosystem market speeds updated
        assertEq(
            state.ecoConfigAfter.supplyEmissionsPerSec,
            state.expectedEcoSpeed,
            "Ecosystem supply speed should match expected"
        );
        assertEq(
            state.ecoConfigAfter.borrowEmissionsPerSec,
            state.ecoConfigBefore.borrowEmissionsPerSec,
            "Ecosystem borrow speed should remain unchanged (supply gauge only)"
        );

        // 4. Core market speeds updated
        assertEq(
            state.coreSupplySpeedAfter,
            state.coreSupplySpeedBefore,
            "Core supply speed should remain unchanged (borrow gauge only)"
        );
        assertEq(
            state.coreBorrowSpeedAfter,
            state.expectedCoreSpeed,
            "Core borrow speed should match expected"
        );
    }

    console2.log("voter.isActive(gEcoSupply): %s", voter.isActive(gEcoSupply));
    console2.log("Ecosystem continues to receive rewards despite the gauge being deactivated");
}
```

Recommended Mitigation: Consider preventing gauges from being deactivated on the voter without also notifying the registrar.

BENQI: Acknowledged.

Cyfrin: Acknowledged.

7.2.5 Speed calculation precision handling is ineffective

Description: `SpeedCalculator::calcSpeed` intends to address precision loss by the application of the `SPEED_PRECISION` constant; however, its presence on both the numerator and denominator does nothing for precision loss as this factor is immediately divided back out:

```
function calcSpeed(
    uint256 _votes,
    uint256 _totalVotes,
    uint256 _moduleBudget,
    uint256 _epochDuration
) public pure returns (uint256) {
    if (_totalVotes == 0 || _epochDuration == 0 || _votes == 0) return 0;

    return
        (_moduleBudget * SPEED_PRECISION * _votes) /
        (_epochDuration * _totalVotes * SPEED_PRECISION);
}
```

It would be better to reverse this scaling when calculating the market budget used, i.e. first performing the speed by duration multiplication. Only then should the result be divided back down, being careful to ensure that all other usage is equivalent as it is currently, e.g. when comparing with `MIN_SPEED` and current speeds on the comptroller/multi-distributor, given that this logic is implemented slightly differently between `BenqiCoreModule` and `BenqiEcosystemModule`.

BENQI: Fixed in PR #16.

Cyfrin: Verified. The use of `SPEED_PRECISION` has been removed entirely, rather than scaling in subsequent usage as recommended.

7.3 Gas Optimization

7.3.1 Avoid initializing variables to default values

Description: There are a number of instances throughout the in-scope contracts where variables are unnecessarily intiialized to default values. For example, consider the following `grep` commands that show the most common scenarios in which this occurs:

```
grep -E 'uint.+ *= *0;' -R --include="*.sol" --exclude-dir=lib .
grep -E 'bool.+ *= *false;' -R --include="*.sol" --exclude-dir=lib .
```

Recommended Mitigation: Avoid initializing variables to default values to save gas.

BENQI: Acknowledged.

Cyfrin: Acknowledged.

7.3.2 `SpeedCalculator::calcSpeed` should return early if `_moduleBudget` is zero

Description: `SpeedCalculator::calcSpeed` currently short circuits if any of the `_totalVotes`, `_epochDuration`, or `_votes` parameters are zero. This is beneficial as it avoids wasting gas on unnecessary computation; however, this validation should also include the `_moduleBudget` as multiplication by zero would similarly result in zero being returned.

Recommended Mitigation:

```
function calcSpeed(
    uint256 _votes,
    uint256 _totalVotes,
    uint256 _moduleBudget,
    uint256 _epochDuration
) public pure returns (uint256) {
-   if (_totalVotes == 0 || _epochDuration == 0 || _votes == 0) return 0;
+   if (_totalVotes == 0 || _epochDuration == 0 || _votes == 0 || _moduleBudget == 0) return 0;

    return
        (_moduleBudget * SPEED_PRECISION * _votes) /
        (_epochDuration * _totalVotes * SPEED_PRECISION);
}
```

BENQI: Fixed in PR #15.

Cyfrin: Verified.

7.3.3 Avoid return statements with named return variables

Description: When declaring named return variables such as in `GaugeRegistrar::registerGauge`, it is not necessary to explicitly execute the return statement and this can be removed to save gas.

Recommended Mitigation:

```
function registerGauge(
    address _qiToken,
    Incentive _incentive,
    address _rewardController,
    string calldata _metadataURI
) external auth(GAUGE_REGISTRAR_ROLE) returns (address gaugeAddress) {
    ...
    // Emit event
    emit GaugeRegistered(gaugeAddress, _qiToken, _incentive, _rewardController);
-
-   return gaugeAddress;
}
```

BENQI: Acknowledged.

Cyfrin: Acknowledged.

7.3.4 Duplicated validation in `DistributionManager::initialize` can be removed

Description: `DistributionManager::initialize` will revert if the `_initialBudgetAllocator` is equal to `address(0)`; however, this logic is not necessary as it is duplicated and already present in `_setBudgetAllocator()`.

```
function initialize(
    ...
    IBudgetAllocator _initialBudgetAllocator,
    ...
) external initializer {
    if (address(_initialBudgetAllocator) == address(0))
        revert InvalidAddress("budgetAllocator");
    ...
    _setBudgetAllocator(_initialBudgetAllocator);
    ...
}

function _setBudgetAllocator(IBudgetAllocator _budgetAllocator) internal {
    if (address(_budgetAllocator) == address(0)) revert InvalidAddress("budgetAllocator");
    ...
}
```

BENQI: Fixed in PR #38.

Cyfrin: Verified.

7.3.5 Action count increment in `BenqiEcosystemModule::_buildActions` can be simplified

Description: `BenqiEcosystemModule::_buildActions` increments the action count when processing indicates that the current gauge should not be skipped:

```
if (!prgv.shouldSkip) {
    actions[actionCount] = Action(_target, 0, prgv.actionData);
    actionCount++;
}
```

However, this logic can be simplified to a single line:

```
if (!prgv.shouldSkip) [actionCount++] = Action(_target, 0, prgv.actionData);
```

BENQI: Acknowledged.

Cyfrin: Acknowledged.