CANTINA

Infrared contracts

Security Review

Cantina Managed review by:

0xrajeev, Lead Security Researcher

Cryptara, Security Researcher

Phaze, Security Researcher

Mario Poneder, Associate Security Researcher

December 26, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Infrared simplifies interacting with Proof of Liquidity with liquid staking products such as iBGT and iBERA.

From Jul 22nd to Aug 9th the Cantina team conducted a review of infrared-contracts on commit hash bfc3d006. The team identified a total of **62** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 13
- Low Risk: 18
- Gas Optimizations: 5
- Informational: 26

3 Findings

3.1 Medium Risk

3.1.1 Dead bribe vaults can be used to prevent poking `VotingEscrow` positions

Severity: Medium Risk

Context: Voter.sol#L214, Voter.sol#L171-L213, Voter.sol#L361-L365

Description: In order to vote in `Voter`, users are first required to create a position in `VotingEscrow` locking their governance tokens for a desired lock duration. During this time, the effective voting power of their position (`balanceOfNFT`) decreases linearly until it reaches zero at the time of their unlock. The voting power also determines the shares and reward allocation in the bribe vaults; and is measured at the time of the vote. The share amount is kept constant per NFT until anyone "pokes" the position thereby updating the amount to reflect the current value as seen in the `VotingEscrow` contract. Other voters are incentivized to poke high value positions reducing the total share and increasing their own share in a bribe vault.

Bribe voting rewards can be disabled for certain staking tokens through governance by calling `Voter.killBribeVault()`. In doing so, any future votes that allocate shares to users for those vaults are denied. This check is performed inside the internal function `_vote()`. Poking a position involves reading a position's current balance and their previous votes; and then casting the same vote with the updated balance, also by calling `_vote()`. Therefore, if a user's vote contains a vote for a disabled vault, that position cannot be poked anymore and their balance won't decrease.

It's possible for a malicious user to exploit this in the following scenario:

1. Alice sees that a governance call is being queued to disabled token XYZ.
2. Alice creates a large locked position.
3. Alice votes for her desired tokens and includes a vote of `1 wei` to token XYZ.
4. The governance proposal passes and token XYZ is disabled.
5. Alice's voting power now remains constant and does not decrease even when the lock expires.

This allows Alice to keep her full voting weight and stake in the bribe vaults while also allowing her to withdraw from the lock at any time.

Impact: High, as this allows for earning an unfair amount of rewards and is effectively bypassing a key component of the `VotingEscrow` system—that locked positions should lose their voting power towards the end of their locks.

Likelihood: Low/medium, as this requires an incoming governance proposal to disable vaults and puts some requirements on capital.

Recommendation: Consider refactoring the code to differentiate the cases of voting and poking and revert when a user votes for a disabled vault and continue/skip the token when the position is being poked.

3.1.2 Acknowledged Medium-Low severity issues from Velodrome security review should be reconsidered

Severity: Medium Risk

Context: Velodrome Spearbit Security Review

Description: Infrared derives its Voting logic from a slightly modified version of Velodrome's corresponding contracts. We identified a total of four applicable Medium and Low severity issues that were reported in Velodrome's Spearbit security review and acknowledged by the Velodrome team without fixing it as recommended. We are highlighting those issues here for awareness and potential reconsideration of a different resolution:

1. Issue 5.3.3 (Medium severity):

   Bribe and fee token emissions can be gamed by users - Voters can continue to earn bribes and fees for their votes that are disproportionately higher compared to their decaying veNFT voting weight if no one pokes their positions.

2. Issue 5.3.9 (Medium severity):

   Inconsistent (sic) between balanceOfNFT, balanceOfNFTAt and _balanceOfNFT functions - The flash-loan protection implemented in `balanceOfNFT()` using `ownershipChange[_tokenId] == block.number` is not consistently applied to `balanceOfNFTAt()` and `_balanceOfNFT` functions, which could allow integrators using those functions to bypass it.

3. Issue 6.4.3 (Low severity):

   Managed NFT can vote more than once per epoch under certain circumstances - Users can bypass the one-vote-per-epoch with Managed NFTs by voting once, withdrawing and redepositing NFTs into Managed NFTs, and voting again.

4. Issue 6.4.8 (Low severity):

   Slightly Reduced Voting Power due to Rounding Error - A fully locked NFT will incur a slight loss of vote weight because of rounding errors.

Recommendation: Consider revisiting the original issues to either acknowledge the risk as-is or potentially evaluate a different resolution as per the original recommendation. It is also worth reevaluating other Voting related issues from the report to understand their implications in the context of Infrared.

3.1.3 Validator cutting board weights can disconnect from onchain voting outcomes

Severity: Medium Risk

Context: Infrared.sol#L826-L833

```
function queueNewCuttingBoard(
    bytes calldata _pubkey,
    uint64 _startBlock,
    IBeraChef.Weight[] calldata _weights
) external onlyKeeper {
    if (!isInfraredValidator(_pubkey)) revert Errors.InvalidValidator();
    chef.queueNewCuttingBoard(_pubkey, _startBlock, _weights);
}
```

Description: `Infrared.queueNewCuttingBoard()` queues new cutting board `_weights` (`BGT` receiver address & share percentage) for an Infrared validator with `_pubkey` and which become active at `_startBlock`. This method, which is only accessible via Infrared's `Keeper` role, is a crucial component of the protocol that facilitates to represent the `IRED` onchain voting distribution outcomes as actual cutting board weights for each Infrared validator.

However, the provided `_weights` data and `_startBlock` in this method has to be fully trusted since it relies on offchain data/analysis by Keeper based on the onchain voting outcomes but without any onchain connection or verification.

Impact: High, because any failure to correctly represent the onchain voting outcomes as cutting board weights leads to losses for specific `BGT` receivers in addition to reputational damage due to deviation from voting outcomes.

Likelihood: Low, because the `Keeper` role is expected to be trusted and the relevant offchain code is assumed to be tested as well as reviewed.

Recommendation: Consider implementing a new contract which establishes a trustless connection among the onchain voting outcomes and the process of queuing new cutting board weights. In case the involved computations are too expensive in terms of gas fees, a `view` method could be implemented which allows anyone to verify the currently queued weights against the voting outcomes.

3.1.4 Key voting functionalities are exposed to access control risk outside governance control

Severity: Medium Risk

Context: IVotingEscrow.sol#L178-L179, VotingEscrow.sol#L81, VotingEscrow.sol#L256-L266, VotingEscrow.sol#L1129-L1132, VotingEscrow.sol#L1235-L1241

Description: Infrared's voting contracts are derived from Velodrome from where it retains the `team` address configuration, which is set to the deployer (originally meant to be the address of Velodrome Team multisig) of `VotingEscrow` contract and is authorized to call `setArtProxy()`, `toggleSplit()` and `setVoterAndDistributor()` functions.

`toggleSplit()` controls the splitting feature of veNFTs. `setVoterAndDistributor()` controls the updating of voter and distributor (who has deposit authorization for managed NFTs) contract addresses. In the context of Infrared, these functionalities are ideally meant to be controlled by Governor. Allowing the deployer to continue managing these functionalities outside governance control is risky because deployer keys may not necessarily have the same security posture as Governor, especially after migration to a token-based governance.

Impact: Low, because key voting functionalities are exposed to access control risk outside governance control.

Likelihood: High, because key voting functionalities `toggleSplit()` and `setVoterAndDistributor()` are always controlled by `team` address in the current implementation.

Recommendation: Consider replacing `team` access control for these functionalities with Infrared Governor.

3.1.5 Claiming auctioned tokens in `BribeCollector` can be front-run

Severity: Medium Risk

Context: BribeCollector.sol#L68-L88

Description: Claiming auctioned tokens in `BribeCollector` can be front-run leading to loss of tokens for later callers.

Bribe tokens that accumulate in the `Infrared` contract can be auctioned off through the `BribeCollector` contract. Tokens can be transferred to the bribe collector contract by calling `Infrared.harvestBribes()`. Thereafter, anyone can call `BribeCollector.claimFees()` to claim the ERC20 bribe tokens in exchange for a given `payoutAmount` of `payoutToken`. In order to do so, the caller must first transfer in the required amount of `payoutToken` and then call `BribeCollector.claimFees()`.

During this process, the call to `BribeCollector.claimFees()` can be front-run and the entire set of ERC20 token bribes can be removed from the `BribeCollector` contract. The effect is that the caller will pay the full amount of `payoutToken` while receiving nothing in return, whereas the front-runner can receive all tokens for free, because they are paying with the caller's tokens. The front-runner can effectively sandwich the calls of payment of `payoutToken` and `BribeCollector.claimFees()`.

Furthermore, it is possible that the `payoutToken` itself is accrued inside the contract if it is part of the bribe or if a previous callers have deposited it and the transaction is being front-run. Any excess amount can be transferred out by specifying the `payoutToken` as a `feeToken`.

Impact: High, as the caller who made the `payoutAmount` of `payoutToken` might lose the expected tokens in return.

Likelihood: Low, as this requires front-running or unfortunate timing and negligence on the caller's side who uses an EOA to call the two functions instead of atomically executing them in a contract call.

Recommendation: Consider adapting the function to include fail-safe mechanisms (illustrated below) such as:

1. Transfer in tokens from caller to make entire claim atomic.
2. Allow the user to specify their expected token amounts and revert if these are not available.
3. Make sure the user is not transferring out the `payoutToken`.
4. Warning users of potential risks when calling `claimFees()`.

Additional recommendations not in example code:

5. Include a call to `Infrared.harvestBribes()` to include pending bribes.
6. Instead of transferring `_feeAmounts`, transfer the ERC20 current balance of `BribeCollector` and use `_feeAmounts` as slippage checks reverting if the transferred balance is lower.

```diff
+ error PayoutTokenClaimed();

- function claimFees(address _recipient, address[] calldata _feeTokens)
-     external
- {
+ function claimFees(
+     address _recipient,
+     address[] calldata _feeTokens,
+     uint256[] calldata _feeAmounts
+ ) external {
+     IERC20(payoutToken).transferFrom(
+         msg.sender, address(this), payoutAmount
+     );
      IERC20(payoutToken).safeIncreaseAllowance(
          address(infrared), payoutAmount
      );

      // Callback into infrared post auction to split amount to vaults and protocol
      infrared.collectBribes(payoutToken, payoutAmount);

      // From all the specified fee tokens, transfer them to the recipient.
      for (uint256 i = 0; i < _feeTokens.length; i++) {
          address feeToken = _feeTokens[i];
-         uint256 feeTokenAmountToTransfer =
-             IERC20(feeToken).balanceOf(address(this));
-         IERC20(feeToken).safeTransfer(_recipient, feeTokenAmountToTransfer);
-         emit FeesClaimed(
-             msg.sender, _recipient, feeToken, feeTokenAmountToTransfer
-         );
+         uint256 feeAmount = _feeAmounts[i];
+         if (feeToken == payoutToken) revert PayoutTokenClaimed();
+         IERC20(feeToken).safeTransfer(_recipient, _feeAmount);
+         emit FeesClaimed(
+             msg.sender, _recipient, feeToken, _feeAmount
+         );
      }
  }
```

3.1.6 Validators can unfairly earn too few or too many rewards

Severity: Medium Risk

Context: Infrared.sol#L723-L762

Description: Validators can be added and removed in `Infrared` which also adds and removes them in `InfraredDistributor` where they receive `ibgt` rewards. When `Infrared.harvestBase()` is called, the accrued rewards are sent as `ibgt` to `InfraredDistributor` through a call to `InfraredDistributor.notifyRewardAmount()`. This immediately distributes the rewards to all current validators.

An issue can arise when rewards accumulate, but have not yet been distributed. When a new validator is added and then `harvestBase()` is called, the new validator receives the same share as previous validators which have been doing more work. Similarly, a validator that has been doing lots of work can be removed before their work is compensated.

Impact: Medium, as validators can miss out on their earned rewards or some can be overly rewarded.

Likelihood: Low/medium, depending on the actual frequency that `harvestBase()` is called with.

Proof of concept: Both Bob and Alice are added as validators at different times.

- `t=0   add(ALICE) amountsCumulativeLast=1.`
- `t=100 add(BOB) amountsCumulativeLast=1.`
- `t=100 notifyRewardAmount(500) amountsCumulative=1 + 250.`

- `t=100 claim(ALICE) amount=1 + 250 - 1.`
- `t=100 claim(BOB) amount=1 + 250 - 1.`

Bob was only recently added as a validator and hasn't done any work yet, however he still receives the same amount of rewards as Alice. Alice's reward is reduced. A similar scenario is possible when removing a validator before calling `harvestBase` making them lose out on any accrued rewards.

- `t=0   add(ALICE) amountsCumulativeLast=1.`
- `t=0   add(BOB) amountsCumulativeLast=1.`
- `t=100 remove(BOB) amountsCumulativeFinal=1.`
- `t=100 notifyRewardAmount(500) amountsCumulative=1 + 500.`
- `t=100 claim(ALICE) amount=1 + 500 - 1.`
- `t=100 claim(BOB) amount=1 - 1.`

Recommendation: Consider changing the function visibility for `harvestBase()` from `external` to `public` and call it before a validator is added/removed in `addValidators()/removeValidators()`.

```diff
- function harvestBase() external whenInitialized {
+ function harvestBase() public whenInitialized {
  // ...
```

```diff
  function addValidators(Validator[] memory _validators)
      external
      onlyGovernor
      whenInitialized
  {
+ harvestBase();
  // ...
```

```diff
  function removeValidators(bytes[] memory _pubkeys)
      external
      onlyGovernor
      whenInitialized
  {
+ harvestBase();
  // ...
```

3.1.7 `Infrared` contract cannot receive bribes in native `BERA`

Severity: Medium Risk

Context: Infrared.sol#L459-L471

Description: According to the `harvestBribes()` method, which forwards bribe assets to the `BribeCollector`, the `Infrared` contract also intends to receive bribes in native `BERA`, which is subsequently wrapped into `WBERA`.

However, `Infrared` contract cannot receive bribes in native `BERA` due to the lack of an external `payable` method or a `receive/fallback` method.

```
function harvestBribes(address[] memory _tokens) external whenInitialized {
    for (uint256 i = 0; i < _tokens.length; i++) {
        address _token = _tokens[i];
        if (_token == DataTypes.NATIVE_ASSET) {
            wbera.deposit{value: address(this).balance}();    // conversion of native BERA to WBERA
            _token = address(wbera);
        }
        // ... skipped for simplicity
    }
}
```

Impact: Medium, because the intended feature to receive native `BERA` bribes is not functional. Furthermore, there might be contract implementations depending on `Infrared` to receive `BERA` as expected.

Likelihood: Medium, although one can alternatively send bribes as tokens to the `Infrared` contract, `BERA` being the native currency can be reasonably expected to be used for a signification portion of the total bribe value over time.

Recommendation: Consider adding a `payable` `receive` method to the `Infrared` contract.

```
receive() external payable {
    // This function is executed when the contract receives native BERA (without data)
}
```

3.1.8 Missing call to `harvestVault()` in `getRewardForUser()` leads to missed rewards

Severity: Medium Risk

Context: MultiRewards.sol#L259-L273, MultiRewards.sol#L280-L287, InfraredVault.sol#L119-L121

Description: Infrared vaults allow stakers to claim rewards either via `getReward()` or via `getRewardForUser(address _user)`. While `getReward()` calls `harvestVault()` in the `onReward()` hook to harvest any BGT rewards from the underlying `BerachainRewardsVault` and corresponding `ibgt ired` token rewards from Infrared, the alternative `getRewardForUser(address _user)` interface is missing a similar call to `harvestVault()` or `onReward()` hook.

Impact: Medium, because any vault stakers attempting to claim rewards via `getRewardForUser(address _user)`, e.g. to allow others claiming on their behalf, will miss out on any rewards from vault harvesting.

Likelihood: Medium, because stakers have an alternative option to claim rewards themselves via `getReward()` to avoid missing rewards.

Recommendation: Consider refactoring `getReward()` and `getRewardForUser(address _user)` so that `harvestVault()` in the `onReward()` hook is appropriately called in both flows.

3.1.9 Reward durations of reward tokens in existing vaults can never be updated

Severity: Medium Risk

Context: InfraredVault.sol#L128-L131, Infrared.sol#L276-L288

Description: While `InfraredVault.updateRewardsDuration()` has a `onlyInfrared` modifier, Infrared does not have any function to call it. Infrared therefore is missing wrapper-function logic to call `updateRewardsDuration()` on any of its existing vaults to update rewards duration of their reward tokens. Whenever Infrared governance updates `rewardsDuration` via `updateRewardsDuration()`, this will only apply to reward tokens of new vaults created or new reward tokens sent.

Impact: Medium, because reward durations of reward tokens in existing vaults can never be updated. This will prevent Infrared governance from effectively adjusting reward rates of existing vaults to enforce fine-grained control on their incentives. Given that enabling incentive flywheels is a primary objective of Infrared, this missing feature will impact the value captured by its vaults.

Likelihood: Medium, because updating reward durations of tokens in existing vaults is anticipated as implemented in Infrared vault's `updateRewardsDuration()` (which has `onlyInfrared` modifier) and `_setRewardsDuration()` logic and as communicated by the project team.

Recommendation: Given that this is an Infrared governance driven action, one option is to enable fine granularity for setting reward durations of tokens in existing Infrared vaults, e.g. for a specific vault+token combination: `function updateRewardsDurationForVaultToken(address _vault, address _rewardsToken, uint256 _rewardsDuration)` or all reward tokens of a specific vault: `function updateRewardsDurationForVault(address _vault, uint256 _rewardsDuration)`.

3.1.10 Griefing attack vector in MultiRewards can lead to zero reward yield due to precision loss

Severity: Medium Risk

Context: MultiRewards.sol#L156-L162, MultiRewards.sol#L76-L88, MultiRewards.sol#L148-L163

Description: The updateReward modfier of the MultiRewards contract updates the rewardPerTokenStored and the associated timestamp lastUpdateTime. Based on these values, the earned rewards of stakers can be computed and subsequently claimed.

```
modifier updateReward(address account) {
    for (uint256 i; i < rewardTokens.length; i++) {
        address token = rewardTokens[i];
        rewardData[token].rewardPerTokenStored = rewardPerToken(token);
        rewardData[token].lastUpdateTime = lastTimeRewardApplicable(token);
        // ... skipped for simplicity
    }
    _;
}

function rewardPerToken(address _rewardsToken)
    public
    view
    returns (uint256)
{
    // ... skipped for simplicity
    return rewardData[_rewardsToken].rewardPerTokenStored.add(
        lastTimeRewardApplicable(_rewardsToken).sub(
            rewardData[_rewardsToken].lastUpdateTime
        ).mul(rewardData[_rewardsToken].rewardRate).mul(1e18).div(    // precision loss
            _totalSupply
        )
    );
}
```

The mentioned rewardPerTokenStored is obtained using the rewardPerToken method. In this method, the computation (lastTimeRewardApplicable - lastUpdateTime) * rewardRate * 1e18 / _totalSupply is subject to a precision loss which causes the residual amount (lastTimeRewardApplicable - lastUpdateTime) * rewardRate * 1e18 % _totalSupply to be unutilized and therefore the associated rewards become stuck in the contract, since only rewards that are accounted in rewardPerTokenStored can be claimed by stakers. As a consequence, these irrecoverable reward amounts accumulate over time within the MultiRewards contract.

However, there is an additional edge-case which allows for a griefing attack leading to zero reward yield of a given _rewardsToken for all stakers. Whenever the condition (lastTimeRewardApplicable - lastUpdateTime) * rewardRate * 1e18 <_totalSupply is satisfied, the value of rewardPerTokenStored will stay unchanged due to precision loss while lastUpdateTime is still updated leading to a full loss of rewards for the duration dt = lastTimeRewardApplicable - lastUpdateTime.

While this attack requires a very high _totalSupply and/or a very low rewardRate to be possible, the dt can be controlled by the attacker and could be a low as 2 seconds (~1.4 seconds block time), since the updateReward modifier can be permissionlessly invoked through the getRewardForUser method. Potentially, this attack could be executed in every block for a whole rewardDuration to cause zero reward yield for stakers.

Note that even if the above condition leading to unchanged rewardPerTokenStored cannot be satisfied, an attacker could still amplify the general precision loss by frequently invoking getRewardForUser and therefore creating a small dt.

Impact: Medium, because in addition to the continuous build-up of small irrecoverable reward amounts, the whole rewards for a given duration could be subject to griefing under certain circumstances.

Likelihood: Medium, although the circumstances for a full griefing attack are rare, there is still a continuous build-up of small irrecoverable reward amounts on every invocation of updateReward for every _rewardsToken.

Proof of concept: The following test case demonstrates the edge-case of a 100% reward loss, i.e. the full reward amount being stuck due to rewardPerTokenStored staying unchanged as a result of griefing.

```
function test_rewardPerTokenStored() public {
    // Setup
    address user1 = address(0xa11ce);

    MockERC20 rewardToken = new MockERC20("", "", 18);
    MockERC20 stakingToken = new MockERC20("", "", 18);

    uint256 rewardDuration = 1 hours;   // short reward duration to save test execution time
    uint256 stakingAmount = 21 ether;
    uint256 rewardRate = 10;

    MockMultiRewards vault = new MockMultiRewards(
        address(stakingToken), address(rewardToken), rewardDuration
    );

    // User1 stakes
    vm.startPrank(user1);
    stakingToken.mint(user1, stakingAmount);
    stakingToken.approve(address(vault), stakingAmount);
    vault.stake(stakingAmount);
    vm.stopPrank();

    // Notify reward
    uint256 rewardAmount = rewardRate * rewardDuration;
    rewardToken.mint(address(this), rewardAmount);
    rewardToken.approve(address(vault), rewardAmount);
    vault.notifyRewardAmount(address(rewardToken), rewardAmount);

    uint256 rewardPerTokenStored;
    uint256 totalSupply = stakingAmount;

    (,,,,, rewardPerTokenStored) = vault.rewardData(address(rewardToken));

    assertEq(rewardPerTokenStored, 0);

    // We can skip time without increasing `rewardPerTokenStored`
    // when `dt * rewardRate * 1e18 < _totalSupply`
    uint256 dt = totalSupply / (rewardRate * 1e18); // dt = 2 (feasible due to ~1.4s block time)

    // Grief rewards for full duration
    uint256 nSkips = rewardDuration / dt;
    for (uint256 i; i < nSkips; i++) {
        skip(dt);

        // Update reward
        vault.getRewardForUser(address(0));
    }

    // User1 is not earning, because `rewardPerTokenStored` is not increasing.
    uint256 lastUpdateTime;
    (,,,, lastUpdateTime, rewardPerTokenStored) =
        vault.rewardData(address(rewardToken));

    assertEq(lastUpdateTime, block.timestamp);
    assertEq(rewardPerTokenStored, 0);

    assertEq(vault.earned(user1, address(rewardToken)), 0);
}
```

Recommendation: The general precision loss can be mitigated by storing the residual amount `(lastTimeRewardApplicable - lastUpdateTime) * rewardRate * 1e18 % _totalSupply` and re-applying it on the next invocation of `updateReward` to prevent the continuous accumulation of irrecoverable amounts.

3.1.11 `InfraredVault` rewards are lost when there are no stakers

Severity: Medium Risk

Context: MultiRewards.sol#L336-L363, MultiRewards.sol#L172-L182

Description: Rewards for staking assets are claimed and transferred by the `Infrared` contract and are accrued in the staking asset's `InfraredVault` to be claimed by its stakers. `Infrared` calls `InfraredVault.notifyRewardAmount` which immediately distributes the `rewardAmount` over the `rewardsDuration` period to all stakers in proportion to their shares.

As there is no logic to pause or retrieve reward emissions when there are no stakers, these end up being stuck inside the vaults.

Impact: High/medium, as reward tokens are lost for the continuous duration when there are no stakers.

Likelihood: Low, as in order for rewards to accrue, there have to have previously been stakers. This issue arises as soon as there are no stakers left and the loss is proportional to the duration in which there are no stakers.

Proof of concept:

```
function test_MultiRewards_notifyReward_noStakers() public {
    // Setup
    address user1 = address(0xa11ce);

    MockERC20 rewardToken = new MockERC20("", "", 18);
    MockERC20 stakingToken = new MockERC20("", "", 18);

    uint256 rewardDuration = 1 weeks;
    uint256 stakingAmount = 100 ether;
    uint256 rewardAmount = 1 ether;

    MockMultiRewards vault = new MockMultiRewards(
        address(stakingToken), address(rewardToken), rewardDuration
    );

    // Notify reward
    rewardToken.mint(address(this), rewardAmount);
    rewardToken.approve(address(vault), rewardAmount);
    vault.notifyRewardAmount(address(rewardToken), rewardAmount);

    skip(rewardDuration);

    // User1 stakes
    vm.startPrank(user1);
    stakingToken.mint(user1, stakingAmount);
    stakingToken.approve(address(vault), stakingAmount);
    vault.stake(stakingAmount);
    vm.stopPrank();

    skip(rewardDuration);

    // The reward is lost and cannot be earned anymore
    assertEq(vault.earned(user1, address(rewardToken)), 0);
}
```

Recommendation: Consider adding a single share (`1 wei`) owned by the `Infrared` contract to `InfraredVault` and include a function allowing governance to claim leftover fees.

3.1.12 Bribe voting rewards that are distributed in epochs with no depositors are lost

Severity: Medium Risk

Context: Reward.sol#L278-L288

Description: Rewards that are sent to `BribeVotingReward` vaults are lost if there are no vault depositors for that epoch.

Anyone can send bribe voting rewards to vaults by calling `BribeVotingReward.notifyRewardAmount()`. This updates the mapping `tokenRewardsPerEpoch[token][epochStart]` which stores the token rewards per epoch with the newly added `amount`. Since the rewards are stored and distributed in their entirety by epochs, they will be lost and irrecoverable if there are no vault depositors for those epochs.

Impact: High, as any amount of reward tokens will be irrecoverably locked.

Likelihood: Low/medium, since this requires someone to send bribe voting rewards to a vault when there are no previous depositors. In order for this issue to arise, an epoch needs to pass where no depositors/voters for that vault remain at the end of the epoch.

Proof of concept:

```
MockERC20 rewardToken = new MockERC20("", "", 18);
MockERC20 stakingToken = new MockERC20("", "", 18);

uint256 constant EPOCH = 1 weeks;

// Mock Voter & VotingEscrow
address public ve;
mapping(uint256 tokenId => address owner) public ownerOf;

function test_BribeVotingReward_notifyReward_noStakers() public {
    // Setup
    address alice = address(0xa11ce);
    uint256 stakingAmount = 100 ether;
    uint256 rewardAmount = 1 ether;

    // Mock mint/access control
    ve = address(this);
    address voter = address(this);
    uint256 tokenId = 1;
    ownerOf[tokenId] = alice;

    address[] memory rewardTokens = new address[](1);
    rewardTokens[0] = address(rewardToken);
    BribeVotingReward vault = new BribeVotingReward(voter, rewardTokens);

    // Notify reward
    rewardToken.mint(address(this), rewardAmount);
    rewardToken.approve(address(vault), rewardAmount);
    vault.notifyRewardAmount(address(rewardToken), rewardAmount);

    // An epoch passes without any stakers
    skip(EPOCH);

    // Deposit `stakingAmount` into reward vault with `tokenId`
    vault._deposit(stakingAmount, tokenId);

    // Another epoch passes with Alice staking
    skip(EPOCH);

    // Alice withdraws
    vault._withdraw(stakingAmount, tokenId);

    // The `rewardAmount` still remains in the vault
    // while Alice has not earned any rewards
    assertEq(rewardToken.balanceOf(address(vault)), rewardAmount);
    assertEq(vault.earned(address(rewardToken), tokenId), 0);

    // Alice stakes again
    vault._deposit(stakingAmount, tokenId);

    // A new reward is notified
    rewardToken.mint(address(this), rewardAmount);
    rewardToken.approve(address(vault), rewardAmount);
    vault.notifyRewardAmount(address(rewardToken), rewardAmount);

    // Skip another epoch
    skip(EPOCH);
    vault._withdraw(stakingAmount, tokenId);

    // Alice has earned `rewardAmount`
    // while the original `rewardAmount` is stuck in the vault
    assertEq(vault.earned(address(rewardToken), tokenId), rewardAmount);
    vault.getReward(tokenId, rewardTokens);
    assertEq(rewardToken.balanceOf(address(vault)), rewardAmount);
    assertEq(rewardToken.balanceOf(address(alice)), rewardAmount);
}
```

Recommendation: Consider adding a function `renotifyRewardAmount` allowing for undistributed rewards to be redistributed.

```
  abstract contract Reward is IReward, ReentrancyGuard {
      constructor(address _voter) {
          voter = _voter;
          ve = IVoter(_voter).ve();
+         // Make initial supply checkpoint explicit in case
+         // rewards are distributed during implicit 0 checkpoint.
+         _writeSupplyCheckpoint();
      }

      // ...

+     error NonZeroSupply();
+     error ActiveEpoch();

+     function renotifyRewardAmount(uint256 timestamp, address token) public {
+         uint256 epochStart = VelodromeTimeLibrary.epochStart(timestamp);
+         uint256 currentEpochStart =
+             VelodromeTimeLibrary.epochStart(block.timestamp);
+         uint256 rewardAmount = tokenRewardsPerEpoch[token][epochStart];
+         uint256 index = getPriorSupplyIndex(timestamp);

+         if (rewardAmount == 0) revert ZeroAmount();
+         if (currentEpochStart <= epochStart) revert ActiveEpoch();
+         if (supplyCheckpoints[index].supply != 0) revert NonZeroSupply();

+         tokenRewardsPerEpoch[token][epochStart] = 0;

+         // Redistribute rewards to current epoch.
+         tokenRewardsPerEpoch[token][currentEpochStart] += rewardAmount;

+         emit NotifyReward(address(this), token, epochStart, rewardAmount);
+     }
  }
```

3.1.13 Reward amounts irrecoverably accumulate in `MultiRewards` due to precision loss

Severity: Medium Risk

Context: MultiRewards.sol#L336-L357

Description: The `_notifyRewardAmount()` method of the `MultiRewards` contract handles an incoming `reward` amount of a `_rewardsToken` by updating the associated `rewardRate`, which facilitates further distribution of the rewards to the stakers.

```
function _notifyRewardAmount(address _rewardsToken, uint256 reward)
    internal
    updateReward(address(0))
{
    // ... skipped for simplicity
    IERC20(_rewardsToken).safeTransferFrom(
        msg.sender, address(this), reward                           // full reward amount pulled in
    );
    if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
        rewardData[_rewardsToken].rewardRate =
            reward.div(rewardData[_rewardsToken].rewardsDuration);    // precision loss
    } else {
        // ... similar computation as above
    }
   // ... skipped for simplicity
}
```

In this method, the computation `rewardRate = reward / rewardsDuration` is subject to a precision loss which causes the residual amount `reward % rewardsDuration` to become stuck in the contract, since the full `reward` amount is transferred in while there is no mechanism to recover or utilize the residual amount.

As a consequence, these irrecoverable reward amounts accumulate over time within the `MultiRewards` contract.

Impact: Low, because the affected reward amounts `reward % rewardsDuration` are small.

Likelihood: High, because the irrecoverable rewards keep accumulating on every invocation of `_notifyRewardAmount` for every `_rewardsToken`.

Proof of concept: The following test case demonstrates the edge-case of a 100% reward loss, i.e. the full `reward` amount being stuck.

```
function test_rewardPerTokenPrecision() public {
    // Setup
    MockERC20 rewardToken = new MockERC20("", "", 18);
    MockERC20 stakingToken = new MockERC20("", "", 18);

    uint256 rewardDuration = 7 days;

    MockMultiRewards vault = new MockMultiRewards(
        address(stakingToken), address(rewardToken), rewardDuration
    );

    // User stakes
    uint256 stakingAmount = 1 ether;
    stakingToken.mint(address(this), stakingAmount);
    stakingToken.approve(address(vault), stakingAmount);
    vault.stake(stakingAmount);

    // Notify reward
    uint256 rewardAmount = rewardDuration - 1; // interpret rewardDuration as amount
    rewardToken.mint(address(this), rewardAmount);
    rewardToken.approve(address(vault), rewardAmount);

    vault.notifyRewardAmount(address(rewardToken), rewardAmount);

    // Percentage loss = (rewardAmount - rewardAmount / rewardDuration * rewardDuration) / rewardAmount
    // Percentage loss = 1 - rewardAmount / rewardDuration / (rewardAmount / rewardDuration)
    // Percentage loss = 1 - rewardRate / rewardRate
    // Percentage loss = 100%
    assertEq(vault.rewardPerToken(address(rewardToken)), 0);
    assertEq(vault.getRewardForDuration(address(rewardToken)), 0);

    skip(rewardDuration);

    uint256 balanceBefore = rewardToken.balanceOf(address(this));
    vault.getReward();
    uint256 balanceAfter = rewardToken.balanceOf(address(this));
    assertEq(balanceAfter - balanceBefore, 0);
}
```

Recommendation: Consider storing the residual amount `reward % rewardsDuration` and add it to the `reward` amount on the next invocation to effectively avoid the accumulation of irrecoverable amounts.

3.2 Low Risk

3.2.1 Burned `ibgt` is reminted as rewards and may lead to double-counting

Severity: Low Risk

Context: Infrared.sol#L410

Description: `harvestBase()` determines the newly accumulated BGT rewards to Infrared by subtracting `ibgt.totalSupply()` from `_bgt.balanceOf(address(this))`. This assumes that all `ibgt` is minted against accumulated BGT rewards within Infrared as part of `_handleBGTRewardsForVault()` and `_handleBGTRewardsForDistributor()`, which is valid.

However, if someone burns their `ibgt` for some reason thus reducing `ibgt.totalSupply()`, then that amount will also be considered as part of the differential BGT to be harvested. But that differential BGT does not correspond to any new BGT rewards. This lets `Distributor` and `wiberaVault` get more `ibgt` reward tokens than that corresponding to BGT actually accumulated in Infrared from Berachain since the previous `harvestBase()`. This effectively makes burning `ibgt` a form of donation to Infrared for reusing towards rewarding.

The project has future plans to allow burning `ibgt` for `bera`. In that case however, any burned `ibgt` will be double-counted towards minting `bera` while also reusing it towards rewarding.

Impact: Low, if the burned `ibgt` is only reused for rewarding. High, if new logic to burn `ibgt` for `bera` is introduced while reusing it towards rewarding.

Likelihood: Low, because there is currently no incentive to burn `ibgt`. Medium, if `ibgt` can be burned for `bera` in future.

Recommendation: The amount of `ibgt` minted in Infrared could be explicitly tracked for determining accumulated BGT rewards instead of relying on `ibgt.totalSupply()`.

3.2.2 Not enforcing an upper threshold on `rewardDuration` can affect vault staking

Severity: Low Risk

Context: Infrared.sol#L194-L195, Infrared.sol#L276-L288

Description: The setting of `rewardDuration` in the constructor and in `updateRewardsDuration()` does not enforce any reasonable upper threshold on the value.

If the deployer/governance makes a mistake and reward duration is set to an unintended high value (effectively causing very low reward rate and therefore low staker incentive), vaults will be created for tokens using that incorrect value. And given the related issue of reward duration of existing vaults unable to be updated, such vaults may lead to staker abandonment. New vaults can never be re-registered for those staking tokens and so those tokens/vaults are effectively excluded from Infrared PoL.

Even if the related issue were to be fixed, vaults with low reward rates in windows of unreasonably high reward durations (while governance is updating the reward duration) will have low staking incentives and may lead to staker abandonment during those periods.

Impact: Low, because if the related issue is fixed, the impact is only limited to windows of unreasonably high reward duration until it is updated.

Likelihood: Low, because it is assumed that parameters in governance proposals are carefully vetted before execution.

Recommendation: Consider enforcing a reasonable upper threshold on `rewardDuration` in constructor and `updateRewardsDuration()`.

3.2.3 Gauges are not checked for being "*friends of the chef*" when registering vaults

Severity: Low Risk

Context: Infrared.sol#L241-L250

Description: When a vault (`InfraredVault`) is registered for a corresponding asset, the asset's gauge (`BerachainRewardsVault`) is never checked whether it is a "friend of the chef". In Berachain, `BGT` tokens are emitted only for gauges that are registered in `BeraChef` (`chef.isFriendofTheChef(gauge)`). If a vault is registered for an asset which is not eligible `BGT` emissions, then users might inadvertently stake in this vault without earning any rewards and vote for it without the vote being able to count.

Impact: The impact is low as no funds are lost, however users might stake tokens without earning rewards or vote for vaults that cannot be included in the final cutting board distribution.

Likelihood: The likelihood is low as Infrared's Keeper contract will likely not create vaults that are not eligible for `BGT` rewards.

Recommendation: Document and make it clear to users that Infrared vaults might not be eligible for any `BGT` rewards. Alternatively, check whether a staking asset's gauge is a "*friend of the chef*" when creating the rewards vault.

```
  function _createRewardsVaultIfNecessary(
      address _infrared,
      address _stakingToken
  ) private returns (IBerachainRewardsVault) {
      IBerachainRewardsVaultFactory rewardsFactory =
          IInfrared(_infrared).rewardsFactory();
      address rewardsVaultAddress = rewardsFactory.getVault(_stakingToken);
      if (rewardsVaultAddress == address(0)) {
          rewardsVaultAddress =
              rewardsFactory.createRewardsVault(_stakingToken);
      }
+     IBeraChef chef = IInfrared(_infrared).chef();
+     if(!chef.isFriendOfTheChef(rewardsVaultAddress) revert VaultNotFriendOfTheChef();
      return IBerachainRewardsVault(rewardsVaultAddress);
  }
```

Note that the function creates a reward vault (gauge) and then checks whether the vault is a "friend of the chef". Vaults can be set to be a "friend of the chef" before creation, as their address are calculated deterministically via `factory.predictRewardsVaultAddress(stakingToken)`.

3.2.4 Activating boosts can be front-run to grief Keeper

Severity: Low Risk

Context: Infrared.sol#L875-L883

Description: When a boost is queued it needs to be activated after a small waiting period. While queueing is a privileged function, activation does not contain any access controls. Infrared manages these calls for validators through a Keeper contract. Boost activation can be done in batch and also does not require any permissions to be called. If a boost has already been activated, calling `activateBoost()` again will make the second transaction revert.

This opens up a small DoS attack vector. If activating boosts is mainly handled by the Keeper, the following scenario is possible:

1. Keeper sends out a transaction activating the boost of 20 validators.
2. Someone front-runs the transaction while only activating the boost of 1 of those validators.
3. Keeper's transaction from (1) reverts because it includes the one already activated validator from (2).
4. Keeper now needs to re-evaluate which validators to boost and re-send a transaction with 19 validators.
5. This transaction can be front-run again creating more work for the Keeper.

Even if this is not done intentionally, it could happen by accident.

Impact: Low, as no real damage can be done aside from being forced to re-send a limited number of transactions and potentially confuse the Keeper.

Likelihood: Low, as this requires unfortunate timing and otherwise there is no real incentive for anyone to do this.

Recommendation: Consider making `activateBoosts` a privileged function only callable by the Keeper by including the `onlyKeeper` modifier. Or use a try-catch statement to allow the transaction to complete when a single activation fails.

```diff
  function activateBoosts(bytes[] memory _pubkeys) external whenInitialized {
      for (uint256 i = 0; i < _pubkeys.length; i++) {
          if (!isInfraredValidator(_pubkeys[i])) {
              revert Errors.InvalidValidator();
          }
-         _bgt.activateBoost(address(this), _pubkeys[i]);
+         try _bgt.activateBoost(address(this), _pubkeys[i]) {
+             // Do nothing if this fails
+         } catch {}
      }
      emit ActivatedBoosts(msg.sender, _pubkeys);
  }
```

3.2.5 Potential Drainage of `payoutToken` in `claimFees` Function

Severity: Low Risk

Context: BribeCollector.sol#L68, BribeCollector.sol#L90

Description: The `claimFees` function in the `BribeCollector` contract does not currently prevent the `payoutToken` from being included in the `_feeTokens` array. This oversight could allow an attacker to drain any remaining `payoutToken` balance by repeatedly calling `claimFees` and including `payoutToken` in the `_feeTokens` array.

Impact: Impact is High because allowing the `payoutToken` to be included in the `_feeTokens` array could result in the depletion of the `payoutToken` balance, leading to a significant loss of funds.

Likelihood: Likelihood is Medium because while it is not typical for `payoutToken` to be included in `_feeTokens`, there is no check preventing it, making it possible for an attacker to exploit this vulnerability.

Recommendation: Add a check to ensure that `payoutToken` is not included in the `_feeTokens` array. This will prevent the `payoutToken` from being drained through the `claimFees` function.

3.2.6 Missing zero address validation in `add()` can cause issues with validator accounting

Severity: Low Risk

Context: InfraredDistributor.sol#L48-L64

Description: The `add()` function in the `InfraredDistributor` contract does not validate the provided validator address. This oversight can lead to scenarios where a zero address is added as a validator, which would be incorrectly counted as a valid validator. This can cause issues with validator management, as the `removeValidators()` function will fail due to the validator being set to the zero address, leading to an inability to remove such validators.

Impact: Medium, because it can lead to incorrect validator management, potentially affecting the overall functionality of the validator system. Adding zero address validators does result in incorrect counts (`numInfraredValidators`) and inability to remove them (`removeValidators`), causing inconsistencies.

Likelihood: Low, because it is unlikely that Governor mistakenly adds a zero address validator. Goverance proposals are assumed to be vetted before execution.

Proof of concept:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.22;

import "forge-std/Test.sol";

import "@core/Infrared.sol";
import "@core/IBGT.sol";
import "@core/InfraredVault.sol";
import "@utils/DataTypes.sol";
import "@utils/ValidatorUtils.sol";
import {IWBERA} from "@berachain/interfaces/IWBERA.sol";

import {ERC1967Proxy} from
    "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

import {IBerachainRewardsVaultFactory} from
    "@berachain/interfaces/IBerachainRewardsVaultFactory.sol";
import {IBeraChef} from "@berachain/interfaces/IBeraChef.sol";
import {BribeCollector} from "@core/BribeCollector.sol";
import {InfraredDistributor} from "@core/InfraredDistributor.sol";
import {Voter} from "@voting/Voter.sol";
import {VotingEscrow} from "@voting/VotingEscrow.sol";

import "@openzeppelin/proxy/Clones.sol";

import "../src/interfaces/IInfrared.sol";

import "@mocks/MockERC20.sol";
import "@mocks/MockWbera.sol";
import "@mocks/MockBerachainRewardsVaultFactory.sol";
import "@mocks/MockBeaconDepositContract.sol";

contract MockBGT is MockERC20 {

    constructor(string memory name, string memory symbol, uint8 decimals)
        MockERC20(name, symbol, decimals)
    {}

    mapping(bytes => uint256) _commissions;

    // COMMISSION_MAX check, not over 224 bits
    function setCommission(bytes memory pubkey, uint256 _newCommissions) public {
        _commissions[pubkey] = _newCommissions;
    }

    // (uint32 blockTimestampLast, uint224 rate)
    function commissions(bytes memory pubkey) public view returns (uint32, uint224) {
        return (0, uint224(_commissions[pubkey]));
    }

}

contract POC is Test {
    using Clones for address;

    IBGT public ibgt;

    MockBGT public bgt;
    MockERC20 public ired;
    MockERC20 public wibera;
    MockERC20 public honey;
    MockWbera public mockWbera;

    MockERC20 public mockPool;

    MockBerachainRewardsVaultFactory public rewardsFactory;
    IBeraChef public beraChef;

    BribeCollector public collector;
    InfraredDistributor public distributor;
    Infrared public infrared;

    Voter public voter;
    VotingEscrow public veIRED;

    uint256 public bribeCollectorPayoutAmount = 10 ether;

    address public admin;
    address public keeper;
    address public governance;

    address stakingAsset;
address public chef = makeAddr("chef"); // TODO: fix with mock chef

function setupProxy(address implementation)
    internal
    returns (address proxy)
{
    proxy = address(new ERC1967Proxy(implementation, ""));
}

function setUp() public {
    // Mock non transferable token BGT token
    bgt = new MockBGT("BGT", "BGT", 18);
    // Mock contract instantiations
    ibgt = new IBGT(address(bgt));
    ired = new MockERC20("IRED", "IRED", 18);
    wibera = new MockERC20("WIBERA", "WIBERA", 18);
    honey = new MockERC20("HONEY", "HONEY", 18);
    mockWbera = new MockWbera();

    // Set up addresses for roles
    admin = address(0x1337);
    keeper = address(1);
    governance = address(2);

    // TODO: mock contracts
    mockPool = new MockERC20("Mock Asset", "MAS", 18);
    stakingAsset = address(mockPool);

    // deploy a rewards vault for IBGT
    rewardsFactory = new MockBerachainRewardsVaultFactory(address(bgt));

    // Set up the chef
    // TODO: fix chef

    // initialize Infrared contracts;
    infrared = Infrared(
        setupProxy(
            address(
                new Infrared(
                    address(ibgt),
                    address(rewardsFactory),
                    address(chef),
                    address(mockWbera),
                    address(honey),
                    address(ired),
                    address(wibera)
                )
            )
        )
    );
    collector = BribeCollector(
        setupProxy(address(new BribeCollector(address(infrared))))
    );
    distributor = InfraredDistributor(
        setupProxy(address(new InfraredDistributor(address(infrared))))
    );

    // IRED voting
    voter = Voter(setupProxy(address(new Voter(address(infrared)))));
    veIRED = new VotingEscrow(
        address(this), address(ired), address(voter), address(infrared)
    );

    collector.initialize(admin, address(mockWbera), 10 ether);
    distributor.initialize();
    infrared.initialize(
        address(admin),
        address(collector),
        address(distributor),
        address(voter),
        1 days
    ); // make helper contract the admin

}
```

```
    function test_add_remove_validator_lock() public {

        IInfrared.Validator memory validator = IInfrared.Validator({
            pubkey: "0x1234",
            addr: address(0),
            commission: 0
        });

        IInfrared.Validator[] memory _validators = new IInfrared.Validator[](1);
        _validators[0] = validator;

        vm.startPrank(admin);
        infrared.addValidators(_validators);

        assertEq(infrared.numInfraredValidators(), 1);

        bytes[] memory _pubkeys = new bytes[](1);
        _pubkeys[0] = "0x1234";

        vm.startPrank(admin);
        infrared.removeValidators(_pubkeys);
    }
}
```

Stack trace:

```
  [180607] POC::test_add_remove_validator_lock()
    [0] VM::startPrank(0x0000000000000000000000000000000000001337)
      ← [Return]
    [158062] ERC1967Proxy::addValidators([Validator({ pubkey: 0x307831323334, addr:
  ↪   0x0000000000000000000000000000000000000000, commission: 0 })])
      [153216] Infrared::addValidators([Validator({ pubkey: 0x307831323334, addr:
  ↪   0x0000000000000000000000000000000000000000, commission: 0 })]) [delegatecall]
        [37130] ERC1967Proxy::add(0x307831323334, 0x0000000000000000000000000000000000000000)
          [32308] InfraredDistributor::add(0x307831323334, 0x0000000000000000000000000000000000000000)
  ↪   [delegatecall]
             emit Added(pubkey: 0x307831323334, validator: 0x0000000000000000000000000000000000000000,
  ↪   amountCumulative: 1)
             ← [Stop]
          ← [Return]
        [3111] MockBGT::commissions(0x307831323334) [staticcall]
          ← [Return] 0, 0
        emit ValidatorCommissionUpdated(_sender: 0x0000000000000000000000000000000000001337, _pubkey:
  ↪   0x307831323334, _current: 0, _new: 0)
        [880] MockBGT::setCommission(0x307831323334, 0)
          ← [Return]
        emit ValidatorsAdded(_sender: 0x0000000000000000000000000000000000001337, _validators: [Validator({
  ↪   pubkey: 0x307831323334, addr: 0x0000000000000000000000000000000000000000, commission: 0 })])
        ← [Return]
      ← [Return]
    [1487] ERC1967Proxy::numInfraredValidators() [staticcall]
      [1180] Infrared::numInfraredValidators() [delegatecall]
        ← [Return] 1
      ← [Return] 1
    [0] VM::startPrank(0x0000000000000000000000000000000000001337)
      ← [Return]
    [6817] ERC1967Proxy::removeValidators([0x307831323334])
      [6485] Infrared::removeValidators([0x307831323334]) [delegatecall]
        [1199] ERC1967Proxy::remove(0x307831323334)
          [879] InfraredDistributor::remove(0x307831323334) [delegatecall]
             ← [Revert] ValidatorDoesNotExist()
          ← [Revert] ValidatorDoesNotExist()
        ← [Revert] ValidatorDoesNotExist()
      ← [Revert] ValidatorDoesNotExist()
    ← [Revert] ValidatorDoesNotExist()
```

Recommendation: Consider adding a validation check to ensure that the validator address is not zero before adding it. This will prevent zero address validators from being added and ensure consistent and correct validator management.

3.2.7 Non-whitelisted reward tokens may be accepted by bribe voting reward vaults

Severity: Low Risk

Context: Voter.sol#L338-L351

Description: `createBribeVault()` does not check if the reward tokens being used during vault creation have been whitelisted by Infrared. While `BribeVotingReward.notifyRewardAmount()` performs a reverting `!IVoter(voter).isWhitelistedToken(token)` check for any new reward tokens being sent, a similar check is missing in `VotingReward.constructor()`.

Impact: Low, because if a Keeper mistakenly includes non-whitelisted reward tokens in `createBribeVault()` they will end up being accepted by that bribe voting reward vault and may cause unexpected token interactions and balances within the protocol.

Likelihood: Low, because Keeper is a trusted role within Infrared and so is expected to include only whitelisted tokens, which means that the likelihood of this happening is low.

Recommendation: Consider adding a reverting `!IVoter(voter).isWhitelistedToken(token)` check for reward tokens being added in `VotingReward.constructor()`. This will prevent any accidental inclusion of such tokens.

3.2.8 Assigning both Keeper and Governance roles to `_admin` is risky

Severity: Low Risk

Context: Infrared.sol#L186-L187

Description: Infrared has two key roles: Governor and Keeper. Governor access controls critical protocol functions and is expected to be initially a multisig with plans to transition to a token-based governance. Keeper access controls operational bot functions such as `registerVault()`, `queueNewCuttingBoard()` and boost functions. However, Infrared currently assigns both `KEEPER_ROLE` and `GOVERNACE_ROLE` to `_admin` during initialization.

The project's plan is to have these roles separated out in the production deployment.

Impact: Medium, because if separation of duties/privileges is not enforced for these two roles in production then any compromise of Keepers can critically impact Governance actions.

Likelihood: Low, because given that project already plans to have these roles separated out in the production deployment, the likelihood of the current super-privileged `_admin` that has both Keeper and Governance roles being deployed in production and being compromised thereafter is presumably very low.

Recommendation: Ensure the separation of these two roles in production deployment.

3.2.9 `BribeVotingReward` does not allow removal of blacklisted reward tokens

Severity: Low Risk

Context: BribeVotingReward.sol#L14-L28

Description: Once the `notifyRewardAmount()` method of `BribeVotingReward` is invoked with a non-reward `token`, its `Infrared` whitelisting status is checked and in case of success it is automatically added as reward token.

```
function notifyRewardAmount(address token, uint256 amount)
    external
    override
    nonReentrant
{
    if (!isReward[token]) {
        if (!IVoter(voter).isWhitelistedToken(token)) {
            revert NotWhitelisted();
        }
        isReward[token] = true;
        rewards.push(token);
    }

    _notifyRewardAmount(msg.sender, token, amount);
}
```

However, `BribeVotingReward` provides no functionality, neither automatically nor manually by governance, to remove a reward token in case it gets removed from `Infrared`'s whitelist, i.e. blacklisted.

Impact: Medium, because the desired mechanism to only allow whitelisted reward tokens is not fully functional and does not protect the contract from operating when whitelisted tokens are removed because they are, for e.g., compromised or deprecated.

Likelihood: Low, because the events which require a token to be removed from the whitelist can be expected to be very rare.

Recommendation: Consider adding a permissionless method which removes a previously whitelisted reward token when it is not whitelisted by `Infrared` anymore.

3.2.10 Irrecoverable dust amounts can accumulate in `InfraredDistributor` due to precision loss

Severity: Low Risk

Context: InfraredDistributor.sol#L98-L110

Description: `InfraredDistributor.notifyRewardAmount()` handles incoming reward `amount` of `IBGT` tokens by increasing `amountsCumulative`. This facilitates further distribution of the rewards among Infrared validators.

```
function notifyRewardAmount(uint256 amount) external {
    // ... skipped for simplicity

    unchecked {
        amountsCumulative += amount / num;     // precision loss
    }
    token.safeTransferFrom(msg.sender, address(this), amount);

    emit Notified(amount, num);
}
```

In this method, the computation `amount / num` is subject to precision loss which causes residual dust amounts `amount % num` to be unclaimable, i.e. irrecoverably stuck in the contract.

Note that the full reward `amount` is transferred in, while there is no mechanism to recover or utilize any unused dust amount.

Impact: Low, because the affected amounts can be considered dust, especially since the `num` of validators is expected to be less initially.

Likelihood: Medium, because the irrecoverable dust amounts keep accumulating on each invocation of `notifyRewardAmount`.

Recommendation: One could keep track of the residual amount `amount % num` and re-apply it on the next invocation of `notifyRewardAmount` to prevent the continuous accumulation of irrecoverable amounts.

3.2.11 Missing `whenInitialized` modifier on functions may cause unexpected behavior

Severity: Low Risk

Context: Infrared.sol#L826-L830, Voter.sol#L118, Voter.sol#L332, BribeCollector.sol#L57

Description: `InfraredUpgradeable` defines a modifier `whenInitialized` which enforces a function to only be callable after the contract has been initialized, i.e. when `initialize()` has been called. While Infrared applies this modifier on many functions, this is missing for `queueNewCuttingBoard()` and few other functions across contracts. This will allow such functions to be called before initialization, which may cause unexpected behavior.

Impact: Calling functions before initialization may cause unexpected behavior.

Likelihood: Upgradeable contracts are expected to be initialized atomically during deployment.

Recommendation: Consider adding `whenInitialized` modifier consistently on all externally visible functions.

3.2.12 Irrecoverable dust amounts can accumulate in `Reward` due to precision loss

Severity: Low Risk

Context: Reward.sol#L174-L219, Reward.sol#L278-L288

Description: `Reward._notifyRewardAmount()` handles an incoming reward `amount` of a `token` for the current epoch by updating the associated `tokenRewardsPerEpoch` mapping. This facilitates further distribution of rewards to the users and the corresponding claimable balance is determined by the `earned` method.

```
function earned(address token, uint256 tokenId)
    public
    view
    returns (uint256)
{
    // ... skipped for simplicity

    if (numEpochs > 0) {
        for (uint256 i = 0; i < numEpochs; i++) {
            // .. skipped for simplicity
            cp0 = checkpoints[tokenId][_index];
            // .. skipped for simplicity
            reward += (cp0.balanceOf * tokenRewardsPerEpoch[token][_currTs])
                / _supply;     // precision loss
            _currTs += DURATION;
        }
    }

    return reward;
}

function _notifyRewardAmount(address sender, address token, uint256 amount)
    internal
{
    if (amount == 0) revert ZeroAmount();
    IERC20(token).safeTransferFrom(sender, address(this), amount);     // full reward amount pulled in

    uint256 epochStart = VelodromeTimeLibrary.epochStart(block.timestamp);
    tokenRewardsPerEpoch[token][epochStart] += amount;

    emit NotifyReward(sender, token, epochStart, amount);
}
```

In the `earned` method, the computation `(cp0.balanceOf * tokenRewardsPerEpoch[token][_currTs]) / _supply` is subject to precision loss which causes residual dust amounts of `tokenRewardsPerEpoch` to be unclaimable, i.e. irrecoverably stuck in the contract.

Note that the full reward `amount` is transferred in, see `_notifyRewardAmount` method, while there is no mechanism to recover or utilize any unused dust amounts.

Impact: Low, because the affected amounts can be considered dust.

Likelihood: Medium, because the irrecoverable dust amounts keep accumulating per user, per epoch and per reward `token`.

Recommendation: One could keep track of unclaimable amounts by storing the residual amounts `(cp0.balanceOf * tokenRewardsPerEpoch[token][_currTs]) % _supply` and allow to recover the associated rewards once the accumulated dust becomes non-negligible.

3.2.13 Voting may be allowed on paused Infrared vaults

Severity: Low Risk

Context: Voter.sol#L191-L213, Infrared.sol#L301-L307

Description: `Voter.sol` manages voting for Infrared's POL `CuttingBoard` allocation of its vaults. Infrared vaults can be paused/unpaused by Infrared Governor to react to any emergency situations. Similarly, their corresponding Voter bribe vaults can be killed/revived by Infrared Governor to allow voting on their cutting board allocations.

While `vote()` prevents voting on killed vaults via a `!isAlive[_stakingToken])` check, it assumes that paused Infrared vaults also have their corresponding bribe vaults killed by Governor.

Impact: If Governor misses killing bribe vaults corresponding to any paused Infrared vaults, voting will be allowed to include such bribe vaults on the cutting board distributions. This may increase POL emissions to paused vaults which may be undesirable depending on the emergency situation that triggered their pausing.

Likelihood: Governor is expected to keep in-sync the pausing/unpausing of Infrared vaults and killing/reviving of their corresponding vote bribe vaults. However, missing to do so is possible and not enforced.

Recommendation: Consider adding a reverting check for `infrared.vaultRegistry(_stakingToken)).paused()` in `_vote()` along with the `!isAlive[_stakingToken])` check.

3.2.14 Functions with `whenInitialized` modifier can be called before initialization

Severity: Low Risk

Context: InfraredUpgradeable.sol#L47-L50

Description: `InfraredUpgradeable` defines a modifier `whenInitialized` which is meant to only be callable when the contract has already been initialized (when `initiailize` has been called). This modifier reverts in case the contract is initializing by checking the internal function `_isInitializing()`.

The `_isInitializing()` function only returns `true` during a call to `initialize`. Before and after the call, `_isInitializing()` will always return `false`. Therefore the the modifier does not achieve the desired effect of only allowing a call after the contract has been initialized. Instead, functions with this modifier can ALWAYS be called outside of the execution of `initialize`.

Impact: Medium, becaues this could potentially open up unintended attack vectors for protected calls.

Likelihood: Low, because functions that are currently decorated with this modifier also contain further modifiers that only allow those functions to be called when the contract has already been initialized.

Recommendation: Consider implementing the modifier as shown below:

```diff
  modifier whenInitialized() {
-     if (_isInitializing()) revert Errors.NotInitialized();
+     uint64 _version = _getInitializedVersion();
+     if (_version == 0 || _version == type(uint64).max) revert Errors.NotInitialized();
      _;
  }
```

3.2.15 `Infrared` is missing the ability to recover ERC20 tokens sent to vaults

Severity: Low Risk

Context: InfraredVault.sol#L168-L177

Description: `InfraredVault` contains functionality to recover ERC20 tokens that were accidentally sent to it. The modifier `onlyInfrared` only allows this method to be called by the `Infrared` contract. However, `Infrared` does not expose any function that calls `InfraredVault.recoverERC20`, therefore making it impossible to recover ERC20 tokens.

Impact: Medium, as a desired token recovery system is not functional.

Likelihood: Low, as this only applies to ERC20 tokens that have unintentionally made their way to the vaults.

Recommendation: Implement an Infrared function that is callable by governance to allow for the recovery of ERC20 tokens sent to its vaults.

```
function recoverERC20FromVault(
    address _asset,
    address _to,
    address _token,
    uint256 _amount
) external onlyGovernor {
    IInfraredVault vault = vaultRegistry[_asset];
    vault.recoverERC20(_to, _token, _amount);
}
```

3.2.16 Compromised Governor can steal whitelisted tokens from Infrared

Severity: Low Risk

Context: Infrared.sol#L310-L321

Description: `Infrared.recoverERC20()` is meant to recover ERC20 tokens that were accidentally sent to the contract or were not whitelisted. However, the recovery logic does not enforce exclusion of whitelisted tokens which include `ibgt`, `ired`, `honey` and `wbera` as part of the initialization.

Impact: A compromised Governor can steal any whitelisted token (`ibgt`, `ired`, `honey` `wbera` and potentially others whitelisted later) from Infrared via `Infrared.recoverERC20()`. This includes whitelisted reward tokens sent to Infrared as bribes and tokens retained as protocol fees.

Likelihood: Governor is initially meant to be a protocol-controlled multisig with a planned transition to a token-controlled governance. So the likelihood of a malfunctioning/compromised Governor accidentally/maliciously triggering a recovery of whitelisted tokens due to compromised multisig or a malicious governance proposal is low.

Recommendation: Exclude whitelisted reward tokens from being recovered via `Infrared.recoverERC20()`.

3.2.17 Calling `Infrared.notifyRewardAmount()` with negligible amounts can delay reward emissions

Severity: Low Risk

Context: MultiRewards.sol#L336-L363

Description: When `InfraredVault.notifyRewardAmount()` is called, any leftover rewards are accumulated and redistributed over the full next `rewardDuration`.

```
function _notifyRewardAmount(address _rewardsToken, uint256 reward)
    internal
    updateReward(address(0))
{
    // handle the transfer of reward tokens via `transferFrom` to reduce the number
    // of transactions required and ensure correctness of the reward amount
    IERC20(_rewardsToken).safeTransferFrom(
        msg.sender, address(this), reward
    );

    if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
        rewardData[_rewardsToken].rewardRate =
            reward.div(rewardData[_rewardsToken].rewardsDuration);
    } else {
        uint256 remaining =
            rewardData[_rewardsToken].periodFinish.sub(block.timestamp);
        uint256 leftover =
            remaining.mul(rewardData[_rewardsToken].rewardRate);
        rewardData[_rewardsToken].rewardRate = reward.add(leftover).div(
            rewardData[_rewardsToken].rewardsDuration
        );
    }

    // ...
}
```

An issue arises when this function can be called by anyone with minimal reward amounts as extending the reward period can effectively delay reward emissions.

- Scenario 1:
  - `t=0`: `notifyReward(100 ether)`.
  - `t=2`: full amount is distributed.
- Scenario 2:
  - `t=0`: `notifyReward(100 ether)`.
  - `t=1`: `notifyReward(0 ether)` (or negligible amounts).
  - `t=2`: 3/4 amount is distributed.

In the second scenario, the distribution rate at is reduced by half at `t=1` extending the total distribution period by 50%.

Impact: Low, because reward amounts are not lost, but are only delayed for an extended period. Infrared expects to use small reward durations not exceeding one week. The impact is therefore further minimized.

Likelihood: High, because `Infrared.harvestVault` can be called by anyone, and even calling this function unintentionally with small reward amounts will delay the full reward emission.

Proof of concept:

```
address alice = address(0xa11ce);

MockERC20 rewardToken = new MockERC20("", "", 18);
MockERC20 stakingToken = new MockERC20("", "", 18);

uint256 rewardDuration = 365 days;
uint256 stakingAmount = 1 ether;
uint256 rewardAmount = 1 ether;

MockMultiRewards vault = new MockMultiRewards(
    address(stakingToken), address(rewardToken), rewardDuration
);

function test_MultiRewards_notifyReward_delay() public {
    // Alice stakes
    vm.startPrank(alice);
    stakingToken.mint(alice, stakingAmount);
    stakingToken.approve(address(vault), stakingAmount);
    vault.stake(stakingAmount);
    vm.stopPrank();

    // Notify reward
    rewardToken.mint(address(this), rewardAmount);
    rewardToken.approve(address(vault), rewardAmount);
    vault.notifyRewardAmount(address(rewardToken), rewardAmount);

    uint256 rewardPeriodStart = block.timestamp;
    uint256 rewardPeriodEnd = block.timestamp + rewardDuration;

    uint256 rewardRate = rewardAmount / rewardDuration;
    uint256 rewardPerToken =
        rewardRate * rewardDuration * 1e18 / stakingAmount;
    uint256 effectiveRewardAmount = stakingAmount * rewardPerToken / 1e18;

    // After the `rewardDuration` passes,
    vm.warp(rewardPeriodEnd);
    // Alice has earned the effective `rewardAmount`
    assertEq(
        vault.earned(alice, address(rewardToken)), effectiveRewardAmount
    );

    // Warp back in time to `t=0`
    vm.warp(rewardPeriodStart);
    assertEq(vault.earned(alice, address(rewardToken)), 0);

    // An attack to delay reward emissions is underway
    uint256 nSkips = 6;
    uint256 dt = rewardDuration / nSkips;
    uint256 rewardPerTokenAcc;
    for (uint256 i; i < nSkips; i++) {
        skip(dt);

        vault.notifyRewardAmount(address(rewardToken), 0);

        rewardPerTokenAcc += rewardRate * dt * 1e18 / stakingAmount;
        rewardAmount = rewardRate * (rewardDuration - dt);
        rewardRate = rewardAmount / rewardDuration;
    }

    uint256 totalEarned = stakingAmount * rewardPerTokenAcc / 1e18;

    // We have reached the end of the reward period,
    // the same time as before, but Alice has earned less
    assertEq(block.timestamp, rewardPeriodEnd);
    assertEq(vault.earned(alice, address(rewardToken)), totalEarned);
    assertLt(totalEarned, effectiveRewardAmount);
    // At about 2/3 of total emissions
    assertApproxEqRel(totalEarned, effectiveRewardAmount * 2 / 3, 0.01e18);

    // Skipping `rewardDuration` is required for the full emissions
    skip(rewardDuration);
    assertApproxEqRel(
        vault.earned(alice, address(rewardToken)),
        effectiveRewardAmount,
        0.00001e18
    );
}
```

Recommendation: Given the minimized impact for low reward durations, the best course of action might be to simply document this possibility and make users aware that the full emission of rewards might be longer than stated.

3.2.18 Missing `version` in EIP-712 domain type hash declaration causes unverifiable `delegateBySig` signatures

Severity: Low Risk

Context: VotingEscrow.sol#L1392-L1400, VotingEscrow.sol#L1253-L1256

Description: The `VotingEscrow` contract does not follow EIP-712's hashing methodology for typed structured data due to a missing `version` parameter in the domain type hash declaration leading to unverifiable EIP-712 signatures when calling `delegateBySig`.

`VotingEscrow`'s `delegateBySig` function allows for voting delegation to other NFT positions. It follows EIP-712's hashing method to verify proper authorization by signatures.

`VotingEscrow` EIP-712 domain type hash is defined using the parameters `string name`, `uint256 chainId` and `address verifyingContract`.

```
bytes32 public constant DOMAIN_TYPEHASH = keccak256(
    "EIP712Domain(string name,uint256 chainId,address verifyingContract)"
);
```

When verifying an EIP-712 structured type hash signature, the domain separator is built using an encoding of the previous domain type hash and the parameters `string name`, `string version`, `uint256 chainId` and `address verifyingContract`.

```
bytes32 domainSeparator = keccak256(
    abi.encode(
        DOMAIN_TYPEHASH,
        keccak256(bytes(name)),
        keccak256(bytes(version)),
        block.chainid,
        address(this)
    )
);
```

The `version` parameter is optional per EIP-712 spec.

> `domainSeparator = hashStruct(eip712Domain)` where the type of `eip712Domain` is a struct named `EIP712Domain` with one or more of the below fields. Protocol designers only need to include the fields that make sense for their signing domain. Unused fields are left out of the struct type.

Here, the `version` parameter is included as part of the encoded data, but not in the type hash definition of the EIP-712 domain. The result is that EIP-712 signatures will not be verifiable according to the spec and integration with wallet apps will be affected. It will not be possible to present structured data to the user to sign. It is still be possible, to create valid signatures when signing over raw data, however, the signer will likely not know what they are signing.

Impact: Low/medium, because the functionality of delegating via signatures is lost.

Likelihood: Low/medium, because this affects delegating via signatures and anyone calling `delegateBySig` with an EIP-712 signature will encounter it. However, normal delegation can still be performed.

Proof of concept:

```
function test_invalidEIP712Signatures() public {
    uint256 pk = uint256(keccak256("Alice"));
    address alice = vm.addr(pk);
    uint256 tokenId1 = createLock(alice);
    uint256 tokenId2 = createLock(user2);

    // Lock permanently to allow delegation
    vm.prank(alice);
    escrow.lockPermanent(tokenId1);

    bytes32 EIP712DomainTypeHash = keccak256(
        "EIP712Domain(string name,uint256 chainId,address verifyingContract)"
    );
    assertEq(escrow.DOMAIN_TYPEHASH(), EIP712DomainTypeHash);

    // Build domain separator according to EIP-712 spec.
    bytes32 domainSeparator = keccak256(
        abi.encode(
            EIP712DomainTypeHash,
            keccak256(bytes("veNFT")),
            block.chainid,
            address(escrow)
        )
    );

    bytes32 structHash = keccak256(
        abi.encode(
            escrow.DELEGATION_TYPEHASH(),
            tokenId1,
            tokenId2,
            0,
            block.timestamp
        )
    );
    bytes32 digest =
        keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(pk, digest);

    vm.expectRevert();
    escrow.delegateBySig(tokenId1, tokenId2, 0, block.timestamp, v, r, s);

    // Rebuild domain separator not conforming to EIP-712 spec.
    domainSeparator = keccak256(
        abi.encode(
            EIP712DomainTypeHash,
            keccak256(bytes("veNFT")),
            keccak256(bytes("2.0.0")),
            block.chainid,
            address(escrow)
        )
    );
    digest =
        keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
    (v, r, s) = vm.sign(pk, digest);

    escrow.delegateBySig(tokenId1, tokenId2, 0, block.timestamp, v, r, s);
}
```

Recommendation: Consider including the `version` parameter in the EIP-712 domain type hash declaration.

```diff
  bytes32 public constant DOMAIN_TYPEHASH = keccak256(
-     "EIP712Domain(string name,uint256 chainId,address verifyingContract)"
+     "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
  );
```

3.3 Gas Optimization

3.3.1 `_stakingTokenVote` can have `memory` attribute instead of `storage`

Severity: Gas Optimization

Context: Voter.sol#L139-L167

Description: `_reset()` uses `address[] storage _stakingTokenVote = stakingTokenVote[_tokenId];` to access `_stakingToken`'s within the loop as part of the reset logic. Given that it uses `delete stakingTokenVote[_tokenId];` directly on the state variable, the `_stakingTokenVote` variable can have `memory` attribute instead of `storage`. This will save several SLOADs in the loop for `_reset()`, which is called in all voting flows.

Recommendation: Consider `_stakingTokenVote` to use `memory` attribute instead of `storage`. Given that this is part of the codebase derived from Velodrome, it is reasonable from a security-perspective to leave it as-is with the understanding that Velodrome is deployed on Optimism which does not have the same gas cost concerns as Berachain.

3.3.2 `rewardTokens.length` can be cached

Severity: Gas Optimization

Context: MultiRewards.sol#L77

Description: `for (uint256 i; i < rewardTokens.length; i++)` repeatedly loads storage variable `rewardTokens.length` in the `updateReward` modifier, which is executed while staking, withdrawing and during reward notifications.

Recommendation: Consider caching `rewardTokens.length` before the loop to avoid repeated `SLOAD`s.

3.3.3 Unoptimized Owner Lookup in `approve` Function

Severity: Gas Optimization

Context: VotingEscrow.sol#L357

Description: In the `approve` function of the `VotingEscrow` contract, the owner of a token is looked up using the `_ownerOf(_tokenId)` function call multiple times. This is inefficient because the owner is already retrieved and stored in the `owner` variable at the beginning of the function. Repeatedly calling `_ownerOf` consumes unnecessary gas.

Recommendation: Refactor the `approve` function to use the pre-computed `owner` variable instead of calling `_ownerOf` multiple times. This will reduce gas consumption and improve the function's efficiency.

Replace:

```
if (_ownerOf(_tokenId) != _from) revert NotOwner();
```

With:

```
if (owner != _from) revert NotOwner();
```

3.3.4 Unnecessary Storage Usage for Interface Support Checks

Severity: Gas Optimization

Context: VotingEscrow.sol#L88-L92, VotingEscrow.sol#L463

Description: The `VotingEscrow` contract uses a mapping (`supportedInterfaces`) to store supported interface IDs. This approach is gas and storage inefficient because the list of supported interfaces is fixed and doesn't change over time. Each entry in the mapping consumes storage and requires gas to read, which increases the overall cost of contract deployment and usage.

Recommendation: Refactor the `supportsInterface` function to use a conditional tree instead of a storage-based approach. This change will eliminate the need for storage and reduce gas costs for checking supported interfaces. Here is a suggested implementation:

```
function supportsInterface(bytes4 interfaceId) public view virtual override returns (bool) {
    return interfaceId == ERC165_INTERFACE_ID || interfaceId == ERC721_INTERFACE_ID || interfaceId ==
↪   ERC721_METADATA_INTERFACE_ID ||
           interfaceId == ERC4906_INTERFACE_ID || interfaceId == ERC6372_INTERFACE_ID;
}
```

3.3.5 Potential inefficiency due to duplicate token addresses in `harvestBribes` Function

Severity: Gas Optimization

Context: Infrared.sol#L459

Description: The `harvestBribes` function does not check for duplicate token addresses in the `_tokens` array. While this does not affect the correctness of the function, it can lead to increased gas usage due to redundant calculations and transfers. The `_handleTokenBribesForReceiver` function calculates the difference between the Infrared contract's token balance and the stored `protocolFeeAmounts` value, transferring the difference. If a token address is duplicated, the first handling adjusts the balance, causing the second handling for the same token to result in zero difference, making `_handleTokenBribesForReceiver` return early.

Recommendation: Add a check to ensure that token addresses in the `_tokens` array are unique before processing them. This can be done using a mapping to track seen addresses and avoid handling duplicates.

3.4 Informational

3.4.1 Missing array lengths mismatch check in `claimBribes()` may lead to unexpected behavior

Severity: Informational

Context: Voter.sol#L380-L392

Description: `claimBribes()` accepts two array parameters `_bribes` and `_tokens` but does not check if their lengths are the same.

Recommendation: Consider adding a array length mismatch check.

3.4.2 Voting features/flows derived from Velodrome should be evaluated

Severity: Informational

Context: Voter.sol, VotingEscrow.sol, BribeVotingReward.sol, VotingReward.sol, Reward.sol

Description: The Voting aspects of Infrared are derived from the Velodrome implementation. Except for the changes introduced, e.g. Bribe vaults and single Fee vault, all other voting and associated reward features/flows are inherited as-is. It is possible that some of this may not be necessary in the Infrared context and may unintentionally increase the attack surface exposed. For e.g., it is not clear exactly if/how the `isWhitelistedNFT` or `managedNFT` features will be used by Infrared.

Recommendation: Consider evaluating if all Voting features/flows derived from Velodrome are necessary. At the same time, it should be acknowledged that modifying a forked codebase has security risks.

3.4.3 Missing event emit in privileged functions

Severity: Informational

Context: Voter.sol#L118-L124, VotingEscrow.sol#L1129-L1132, VotingEscrow.sol#L1235-L1241

Description: `setMaxVotingNum()` is missing an event emit. If governance were to reduce `maxVotingNum`, which is the most number of staking tokens a voter can vote for at once, then votes with `stakingTokenVote.length > maxVotingNum` will revert. Emitting an event in `setMaxVotingNum()` will let voters know the new value of `maxVotingNum` and vote accordingly.

`toggleSplit()` is missing an event emit to indicate if NFTs are allowed to be split or not.

`setVoterAndDistributor()` is missing an event emit to indicate that `voter` and `distributor` addresses have been updated.

Recommendation: Consider emitting events in privileged functions for enhanced transparency.

3.4.4 `Infrared` has to persistently maintain the operator role for each validator on `BeaconDeposit` level

Severity: Informational

Context: Infrared.sol#L813, Infrared.sol#L832

Description: The `Infrared` contract invokes the `BGT.setCommission` and `BeraChef.queueNewCuttingBoard` methods for its validators, which both require the `msg.sender`, i.e. `Infrared`, to be set as the validator's operator on `BeaconDeposit` level, see `IBeaconDeposit.getOperator`.

When adding a validator to Infrared via the `addValidators` method, this operator role is implicitly checked, otherwise the subsequent call to `_updateValidatorCommission` would fail.

However, in case the `Infrared` contract's operator role for a given validator is revoked during operation, any commission updates (also when replacing & removing validators) and cutting board changes are subject to DoS.

Recommendation: It is recommended to revisit potential measures to ensure that the `Infrared` contract persistently maintains the operator role for all its validators.

3.4.5 Discrepancy between Natspec and implementation affects readability

Severity: Informational

Context: See each particular case below.

Description:

- MultiRewards.sol#L276: The comment "Hook called in getRewardForUser function after updating rewards" is misleading. The hook `onReward` is actually called in the `getReward` function before the `getRewardForUser` call. The updateReward modifier, which updates the reward, is executed within `getRewardForUser`. Update the comment to accurately reflect the function flow. Make sure that any changed natspec does not contradict the fixes on the issue "Missing call to `harvestVault()` in `getRewardForUser()` leads to missed rewards".
- IInfraredDistributor.sol#L66-L67: The comments "Removes a validator from validator set to stop commissions" and "Does remove from registry in case claim after remove" are misleading. The removal from the validator set is not implemented in the `remove` method. Furthermore, the subsequent removal from the registry is not implemented in the `claim` method.

Recommendation: This discrepancy between Natspec comments and actual function implementation could cause confusion for developers and auditors. We recommend to update the comments to accurately reflect the implementation.

3.4.6 Events are missing indexed attribute

Severity: Informational

Context: MultiRewards.sol#L61-L65

Description: Indexed event fields make them quickly accessible to off-chain tools. While this is applied across most events, few events are missing this `indexed` attribute for address parameters.

Recommendation: Consider evaluating the application of `indexed` keyword with the understanding that this will lead to slightly increased gas usage during event emission.

3.4.7 `InfraredDistributor.notifyRewardAmount()` allows anyone to donate rewards

Severity: Informational

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

Description: `InfraredDistributor.notifyRewardAmount()` is expected to be called only as part of the `Infrared.harvestBase()` flow. However, it is missing an `onlyInfrared` modifier. This allows anyone to donate `ibgt` rewards to validators. While this shouldn't be any issue in particular (see related issue), it may be worth considering locking the functionality to the expected `Infrared.harvestBase()` flow.

Recommendation: Consider adding an `onlyInfrared` modifier to `InfraredDistributor.notifyRewardAmount()`.

3.4.8 Privileged roles and actions lead to centralization risks for users

Severity: Informational

Context: InfraredUpgradeable.sol#L78-L81, Infrared.sol, BribeCollector.sol, Voter.sol, VotingEscrow.sol

Description: There are several `onlyGovernor` and `onlyKeeper` functions across the protocol that affect critical protocol state and semantics. Some examples are highlighted below:

1. Governor can upgrade `Infrared`, `InfraredDistributor`, `BribeCollector` and `Voter` implementations.
2. Governor can call `updateWhiteListedRewardTokens()`, `updateRewardsDuration()`, `updateIredMintRate()`, `pauseVault()`, `recoverERC20()`, `delegateBGT()`, `claimProtocolFees()`, update fee/weight, add/remove/replace validators, `updateValidatorCommission()` in `Infrared.sol`.
3. Governor can call `setPayoutAmount()` in `BribeCollector.sol`.
4. Governor can call `setMaxVotingNum()`, `whitelistNFT()`, `killBribeVault()`, `reviveBribeVault()` in `Voter.sol`.
5. Governor can call `createManagedLockFor()`, `setAllowedManager()`, `setManagedState()` in `VotingEscrow.sol`.
6. Keeper can call `registerVault()`, `queueNewCuttingBoard()`, `queueBoosts()`, `cancelBoosts()`, `dropBoosts()` in `Infrared.sol` and `createBribeVault()` in `Voter.sol`. Keeper is expected to trigger these functions based on offchain data and analysis.

Recommendation: Consider:

1. Documenting all the privileged roles and actions for protocol user awareness.
2. Enforcing strict role-based access control where different privileged roles control different protocol aspects and are backed by different keys to follow separation-of-privileges security design principle.
3. Enforcing reasonable thresholds and checks wherever possible.
4. Emitting events for all privileged actions.
5. Putting privileged actions affecting critical protocol semantics behind timelocks so that users can decide to exit/engage.
6. Following the strictest opsec guidelines for privileged keys e.g. use of reasonable multisig and hardware wallets.

3.4.9 Function/variable names not accurately reflecting their actions/state affects readability

Severity: Informational

Context: Infrared.sol#L301, Infrared.sol#L376-L377, Infrared.sol#L597, Reward.sol#L222, Reward.sol#L235, Voter.sol#L375

Description: Function/variable names should accurately reflect their actions/state so that developers, users and reviewers do not miss/misunderstand their semantics and side-effects.

There are some functions/variables across the codebase whose names do not accurately indicate their entire/correct functionality. For example:

1. `pauseVault()` should be `togglePauseVault()` for the name to be consistent with implemented logic.
2. `_feeTotal` and `_feeProtocol` are fee rates.
3. `_handleBGTRewardsForVault()` also mints `ired` token rewards.
4. `_deposit`/`_withdraw` begin with an underscore although being `external` methods.
5. Getter function `length()` can be renamed to `stakingTokensLength()`.

Recommendation: Consider using function/variable names that accurately reflect their actions/state.

3.4.10 Multiple definitions of voting epoch duration

Severity: Informational

Context: BalanceLogicLibrary.sol#L11, VelodromeTimeLibrary.sol#L5, Voter.sol#L34, VotingEscrow.sol#L586

Description: Multiple parts of the protocol depend on the voting epoch duration being `7 days`. However, the respective constant is separately defined in four instances which leads to incompatibilities in case one instance is accidentally omitted during a potential future change.

Recommendation: Consider relying on one global definition of the epoch duration across the protocol.

3.4.11 Unused event `WhitelistToken`

Severity: Informational

Context: IVoter.sol#L49-L51

Description: The `WhitelistToken` event is defined in `IVoter` but never used, because the token whitelisting is handled via the`Infrared` contract.

```
event WhitelistToken(
    address indexed whitelister, address indexed token, bool indexed _bool
);
```

Recommendation: Remove the event definition from `IVoter`.

3.4.12 Voting weight might not be fully utilized due to precision loss

Severity: Informational

Context: Voter.sol#L215-L216

Description: The `_vote` method of the `Voter` contract votes with a given `_weight` for multiple `_stakingTokenVote` addresses and this voting power is again distributed respectively using `_weights`.

```
function _vote(
    uint256 _tokenId,
    uint256 _weight,
    address[] memory _stakingTokenVote,
    uint256[] memory _weights
) internal {
    // ... skipped for simplicity
    for (uint256 i = 0; i < _stakingTokenCnt; i++) {
        _totalVoteWeight += _weights[i];
    }

    for (uint256 i = 0; i < _stakingTokenCnt; i++) {
        // ... skipped for simplicity

        uint256 _stakingTokenWeight =
            (_weights[i] * _weight) / _totalVoteWeight;    // precision loss
        // ... skipped for simplicity
    }
    // ... skipped for simplicity
}
```

However, the computation `_stakingTokenWeight = (_weights[i] * _weight) / _totalVoteWeight` is subject to precision loss and therefore the sum of all `_stakingTokenWeights` might not fully add up to `_weight`.

Since `_weight` is not further used in the method, this is not an issue from an accounting perspective. Anyways, it can be considered an UX issue since not all of `_weight` will be utilized as expected.

Recommendation: Once could circumvent the present issue by computing the `_stakingTokenWeight` in the last iteration by subtracting the sum of all previous `_stakingTokenWeights` from `_weight`.

3.4.13 Missing reward token checks in `Reward` and `VotingReward`

Severity: Informational

Context: Reward.sol#L255-L268, Reward.sol#L278-L288, VotingReward.sol#L39-L43

Description: `Reward` and `VotingReward` do not adequately enforce checks on whether a token is whitelisted before allowing it to be processed in the `getReward` and `notifyRewardAmount` functions. Specifically:

- `getReward` **function:** A valid `tokenId` can be passed along with a list of tokens that are not whitelisted. This allows users to interact with non-whitelisted tokens, which can lead to unnecessary storage use and potential confusion in the system.
- `notifyRewardAmount` **function:** This function if extended without proper whitelisting check on tokens will allow users to add any token potentially resulting in the contract being populated with unnecessary and unapproved tokens.

Recommendation: Consider implementing strict checks in both the `getReward` and any future/different extensions of `notifyRewardAmount` functions to ensure that only whitelisted tokens are processed. This will prevent unnecessary data from being stored in the contract and ensure that only valid, approved tokens are handled by these functions.

3.4.14 Duplicate reward tokens check missing in `VotingReward` Constructor

Severity: Informational

Context: VotingReward.sol#L11-L18

Description: The constructor of the `VotingReward` contract does not check for duplicate reward tokens in the `_rewards` array. This can lead to the `rewards` array containing duplicate entries, which might not pose a significant practical issue but can lead to unnecessary gas costs and potential confusion.

Recommendation: Update the constructor to check for duplicate entries in the `_rewards` array before adding them to the `rewards` array. This can be done by verifying if `isReward[_rewards[i]]` is already true before setting it and pushing the token to the `rewards` array.

3.4.15 Missing test coverage across multiple contracts

Severity: Informational

Context: See each particular case below

Description: Several critical functions across multiple contracts lack test coverage. These functions include:

- `Infrared.sol`
  - `claimProtocolFees` (lines 357 to 368).
  - `harvestBase` (lines 409 to 434).
  - `harvestBribes`.
  - `collectBribes`.
  - `harvestBoostRewards`.
  - `addValidators`.
  - `removeValidators`.
  - `replaceValidator`.
- `InfraredDistributor.sol`
  - `purge`.
- `BribeCollector.sol`
  - `claimFees`.
- `Voter.sol`
  - `withdrawManaged`.
  - `claimBribes`.
  - `claimFees`.
- `VotingEscrow.sol`
  - `withdrawManaged`.
  - `approve` .
  - `safeTransferFrom`.
  - `_removeTokenFromOwnerList` (else case).
  - `merge`.
  - `split`.
  - `_createSplitNFT`.
- `BribeVotingReward.sol`
  - `notifyRewardAmount` (lines 20 to 24, `!isReward[token]` case).
- `Reward.sol`
  - `_getReward`.

Recommendation: Implement comprehensive test coverage for the identified functions. Ensure that tests cover various edge cases and scenarios, such as:

- Normal and boundary conditions.
- Failure conditions and error handling.
- Correctness of state changes and event emissions.

Regularly review and update tests to align with any changes in the contract logic. This will enhance the robustness of the code and reduce the risk of critical issues going unnoticed.

3.4.16 Unnecessary `_amountsCumulative == 0` checks

Severity: Informational

Context: InfraredDistributor.sol#L72, InfraredDistributor.sol#L86, InfraredDistributor.sol#L118

Description: The `InfraredDistributor` contract contains unnecessary checks for `_amountsCumulative == 0` in the `remove`, `purge`, and `claim` functions. These checks are redundant because the `validator == address(0)` check, which is performed earlier, already ensures that the validator exists. The `_amountsCumulative == 0` condition will never be met if the validator exists.

Recommendation: Remove the unnecessary `_amountsCumulative == 0` checks from the `remove`, `purge`, and `claim` functions to simplify the code and improve efficiency.

3.4.17 Check in `harvestBase()` can be split to revert with correct errors

Severity: Informational

Context: Infrared.sol#L413

Description: The current check `if (bgtBalance <= minted) revert Errors.UnderFlow();` in the `harvestBase` function is incorrect. The intended logic should be to ensure that there is always more BGT than iBGT, allowing equality (bgtBalance == minted) but preventing more iBGT than BGT. The check should be `if (bgtBalance < minted)`, and there should be a separate check for `bgtAmt != 0` to handle cases where no additional IBGT can be minted because BGT and iBGT are equal.

Recommendation: Consider updating the check to `if (bgtBalance < minted)`, and add a subsequent check for `bgtAmt != 0` to handle cases where no additional IBGT can be minted.

3.4.18 Incorrect comment on reward tokens for IBGT vault

Severity: Informational

Context: Infrared.sol#L197-L201

Description: The comment on line 197 in the `initialize` function of the `Infrared` contract incorrectly states that the IBGT vault can have IBGT and IRED rewards. In reality, the IBGT vault can also have Honey (Bera native stablecoin) as a reward token. This incorrect comment can lead to misunderstandings about the functionality and supported reward tokens of the IBGT vault.

Recommendation: Update the comment to accurately reflect that the IBGT vault can have IBGT, IRED, and Honey as reward tokens. Ensure that all comments within the contract correctly describe the associated code behavior to maintain clarity and prevent misunderstandings.

3.4.19 Inconsistent zero-address validation in `Infrared.initialize()`

Severity: Informational

Context: Infrared.sol#L176-L177

Description: In `Infrared.initialize()`, most addresses are checked for zero value, but the `_voter` address is not validated. This inconsistency can lead to potential issues if `_voter` is inadvertently set to the zero address, causing subsequent calls to fail or behave unexpectedly.

Recommendation: Consider adding a check to ensure that the `_voter` address is not zero, consistent with the validation applied to other addresses in the `initialize` function. This will maintain consistency and reduce the risk of inadvertently setting an invalid address.

3.4.20 Role checking logic can be improved

Severity: Informational

Context: InfraredUpgradeable.sol#L60-L71

Description: The current implementation of `_checkRole` in the `InfraredUpgradeable` contract relies on the `_infrared` variable set in the constructor. This approach can lead to potential inconsistencies and confusion as the `Infrared` contract sets it to zero, while `InfraredDistributor`, `Voter`, and `BribeCollector` set it to non-zero values. Overriding `_checkRole` in each derived contract would result in clearer and more maintainable code, ensuring that role checks are consistently handled based on the specific contract's context.

Recommendation: Consider overriding the `_checkRole` method in each derived contract (`InfraredDistributor`, `Voter`, and `BribeCollector`) to directly check roles on the `infrared` contract. The `Infrared` contract need not override `_checkRole`, defaulting to `super._checkRole`. This approach ensures clarity and consistency in role checking logic.

3.4.21 Duplicated zero value check for rewards duration

Severity: Informational

Context: InfraredVault.sol#L133

Description: The `updateRewardsDuration` method in the `InfraredVault` contract contains a check for a zero value of `_rewardsDuration`. This check is redundant because the internal call to `_setRewardsDuration` already includes a similar check to ensure the rewards duration is non-zero. This duplication increases the complexity and size of the contract unnecessarily.

Recommendation: Consider removing the redundant check for zero value in the `updateRewardsDuration` method, relying on the check within `_setRewardsDuration` to handle this validation. This will simplify the code and potentially reduce gas costs.

3.4.22 Different reward durations for reward tokens may improve flexibility

Severity: Informational

Context: InfraredVault.sol#L42

Description: The current constructor of the `InfraredVault` contract sets a single reward duration for all reward tokens. Allowing different reward durations for each reward token can provide greater flexibility and align with the functionality provided by the `updateRewardsDuration()` method, which already supports setting different durations per token.

Recommendation: Consider modifying the constructor to accept an array of reward durations corresponding to each reward token. Ensure that the length of the `_rewardTokens` array matches the length of the `_rewardsDurations` array to maintain consistency. Update the constructor implementation to iterate through each reward token and set its respective duration.

3.4.23 `MAX_NUM_REWARD_TOKENS` value is inconsistent with documentation

Severity: Informational

Context: InfraredVault.sol#L25

Description: The `MAX_NUM_REWARD_TOKENS` constant is set to 10 in the `InfraredVault` contract, which contradicts the Berachain documentation that states there is a limit of 3 incentive assets per gauge (Documentation). This discrepancy can lead to potential issues with compliance and functionality when more than 3 reward tokens are added.

Recommendation: Consider updating the `MAX_NUM_REWARD_TOKENS` constant to 3 to align with the documentation. Additionally, review the contract and associated documentation to ensure consistency in the limits imposed on the number of incentive assets. This change will ensure the contract remains compliant with the stated criteria for incentives and prevents potential issues from exceeding the documented limits.

3.4.24 Unused `_setRewardsDistributor` method and redundant `rewardsDistributor`

Severity: Informational

Context: MultiRewards.sol#L306, MultiRewards.sol#L326

Description: The `_setRewardsDistributor()` method is defined but never used within the `MultiRewards` contract. Additionally, `rewardsDistributor` is used to manage ACL controls for `setRewardsDuration` and `notifyRewardAmount` functions, which are now managed via an internal function with an `onlyInfrared` modifier in the extending contract `InfraredVault`. The `rewardsDistributor` is set within the `_addReward()` method, making the `_setRewardsDistributor` method redundant.

Presence of unused code does not affect the functionality of the contract but adds unnecessary complexity and can lead to confusion.

Recommendation: Consider removing the `_setRewardsDistributor()` method from the `MultiRewards` contract. Additionally, consider removing `rewardsDistributor` from the system as its functionality is already covered by internal methods with `onlyInfrared` modifier. Ensure that all references to `rewardsDistributor` are removed or refactored to maintain code clarity and simplicity.

3.4.25 Incorrect hook call in `getRewardForUser` Function

Severity: Informational

Context: MultiRewards.sol#L276

Description: The comment "Hook called in getRewardForUser function after updating rewards" is misleading. The hook `onReward` is actually called in the `getReward` function before the `getRewardForUser` call. The `updateReward` modifier, which updates the reward, is executed within `getRewardForUser`. This discrepancy between the comment and the actual function flow could cause confusion for developers and auditors.

Recommendation: Update the comment to accurately reflect the function flow. Specifically, change the comment to indicate that the `onReward` hook is called in the `getReward` function before the `getRewardForUser` call. Ensure that all comments within the contract correctly describe the associated code behavior to maintain clarity and prevent misunderstandings.

3.4.26 Unnecessary use of `SafeMath` Library

Severity: Informational

Context: MultiRewards.sol#L13

Description: `SafeMath` library is imported and used in the `MultiRewards` contract. However, the compiler version specified in the contract is 0.8.22, which includes built-in overflow and underflow checks for integer operations. Using `SafeMath` in this context is redundant and adds unnecessary EVM opcodes, potentially increasing gas costs and reducing the overall efficiency of the contract.

Recommendation: Consider removing the import statement and the usage of `SafeMath` throughout the `MultiRewards` contract. The compiler will handle integer overflow and underflow checks natively, resulting in more efficient bytecode and reduced gas costs.